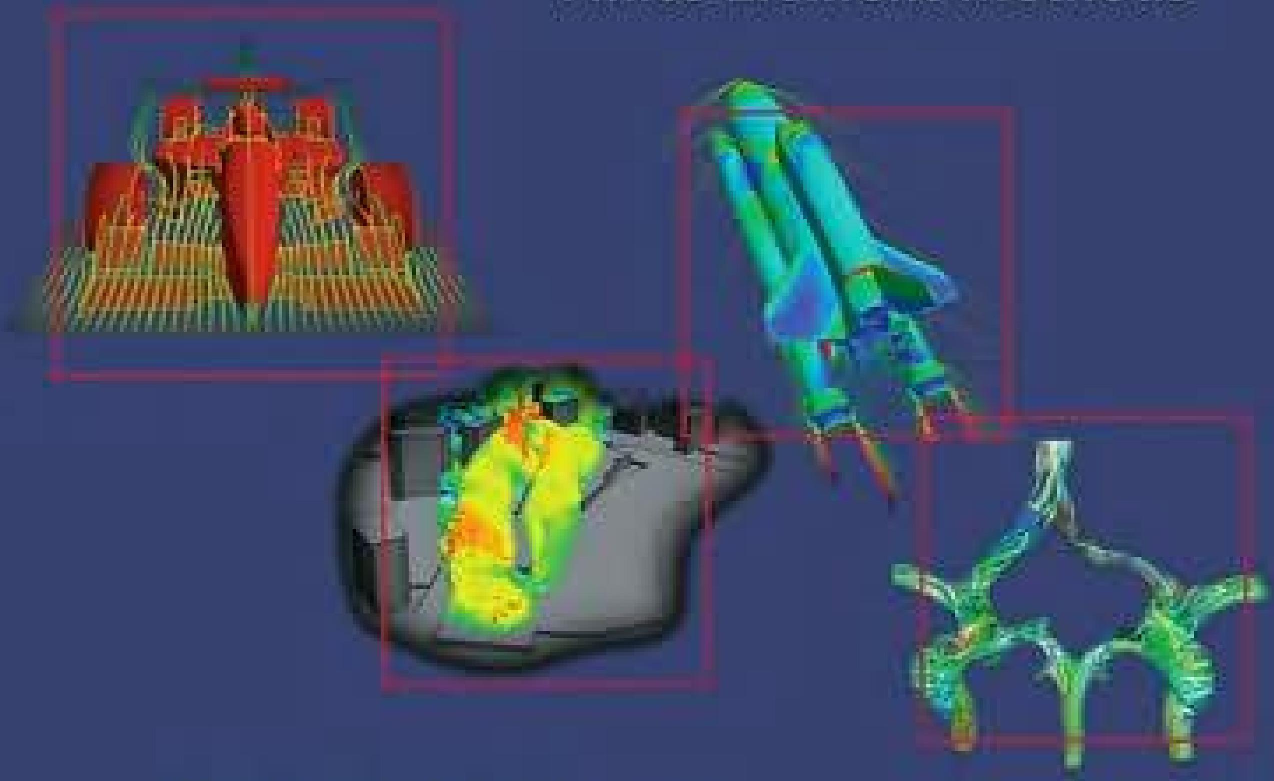
SECOND EDITION

Applied CFD Techniques

An Introduction based on Finite Element Methods



Rainald Löhner

WILEY

APPLIED COMPUTATIONAL FLUID DYNAMICS TECHNIQUES

Applied Computational Fluid Dynamics Techniques: An Introduction Based on Finite Element Methods, Second Edition.
Rainald Löhner © 2008 John Wiley & Sons, Ltd. ISBN: 978-0-470-51907-3

APPLIED COMPUTATIONAL FLUID DYNAMICS TECHNIQUES

AN INTRODUCTION BASED ON FINITE ELEMENT METHODS

Second Edition

Rainald Löhner

Center for Computational Fluid Dynamics,
Department of Computational and Data Sciences,
College of Sciences, George Mason University,
Fairfax, Virginia, USA

John Wiley & Sons, Ltd

Library of Congress Cataloging-in-Publication Data
Löhner, Rainald.
Applied computational fluid dynamics techniques : an introduction based on finite element methods / Rainald Lohner. – 2nd ed.
p. cm.
Includes bibliographical references and index.
ISBN 978-0-470-51907-3 (cloth : alk. paper)
1. Fluid dynamics–Mathematics. 2. Numerical analysis. 3. Finite element method. I. Title.
TA357.L592 2008
620.1’064–dc22
2007045555

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library

ISBN 978-0-470-51907-3

Typeset by Sunrise Setting Ltd, Torquay, UK
Printed and bound in Great Britain by Antony Rowe Ltd, Chippenham, Wiltshire

This book is printed on acid-free paper responsibly manufactured from sustainable forestry in which at least two trees are planted for each one used for paper production.

CONTENTS

FOREWORD TO THE SECOND EDITION

It has now been more than six years since the first edition of this book. Many readers have pointed out errors as well as pedagogical flaws that were present. The author is most grateful for these comments, and hopes the second edition will mitigate most of these shortcomings.

CFD as a field has seen many significant innovations in these few years, some of which have been incorporated throughout the book, as well as new chapters that were not present in the first edition.

Drs Romain Aubry and Fernando Mut were kind enough to read this new edition, providing many comments and suggestions.

ACKNOWLEDGEMENTS

This monograph has its roots in several short courses that were held at the Boeing company, Seattle, in 1988, the IBM Short Course in computational fluid dynamics (CFD), held in Monterey, CA, in 1990, and the AGARD Special Course on Unstructured Grid Methods for Advection Dominated Flows, held at the von Karman Institute in Brussels and the NASA Ames Research Center in 1992. A considerable amount of material has been taken from the yearly advanced graduate studies CFD I,II course taught by the author at George Mason University over the last decade. Moreover, large portions of this text were taken from the author's publications in scientific journals, books and conference proceedings. In much the same way as object-oriented programming, the use of computers has made it a simple matter to compile and edit the material from these publications.

The aim of this book is to provide an introduction to the techniques used in applied CFD. No attempt has been made to provide a comprehensive treatise of all possible methods and algorithms. Given the high rate of innovations and the constant stream of new ideas and publications, such an undertaking would be imprudent at the present time. The emphasis is placed on well-established techniques that have proven their worth in practical applications. In an era that seems more concerned with originality than quality and reliability, this emphasis seems more than justified.

It is my great pleasure to acknowledge the input and stimulus provided by the many colleagues with whom I had the honour to work over the years: from my university team, Drs Jean Cabello, Dorothée Martin, Helen Rudd, Benoît Petitjean, Eric Mestreau, Jean Favre, Alexander Shostko, Chi Yang, Juan Cebral, Makoto Nagaoka, Eric Darve, Jarek Tuzsinsky, Fernando Camelli, Jacob Waltz, Orlando Soto, Marcelo Castro, Joaquin Arteaga-Gomez, Fumiya Togashi, Romain Aubry, Fernando Mut and Sunil Appanaboyina; from the Naval Research Laboratory/Berkeley Research Associates/SAIC/NASA-GSFC teams, Drs Steven Zalesak, Joseph Baum, Jay Boris, David Book, Richard DeVore, John Ambrosiano, Gopal Patnaik, Ravi Ramamurti, Eric Loth, Hong Luo and Dmitri Sharov; from the NASA LARC/Vigyan team, Drs Manuel Salas, Clyde Gumbert, Paresh Parikh and Shaiar Prizadeh; from the Swansea/Imperial College/MIT Teams, Professors Olgierd Zienkiewicz, Kenneth Morgan, Jaime Peraire and Mehdi Vahdati; from the INOVA Fairfax Hospital, Drs James Burgess and Christopher Putnam; from the CIMNE/UPC in Barcelona, Professors Eugenio Oñate, Sergio Idelsohn, Ramon Ribo, Julio Garcia and Carlos Sacco; from the TU Braunschweig, Drs Elmar Walhorn, Björn B. Hübner and Professor Dieter Dinkler.

The work compiled here would not have been possible without the steady support received by the author and his colleagues from such organizations as the Air Force Office of Scientific Research, the Defense Nuclear Agency, the Defense Advanced Research Projects Agency, NASA and the Office of Naval Research. It is my hope that we have served taxpayers' money well by developing the techniques described. It takes years to develop a new field. These organizations have shown time and again that they are willing to be patient and optimistic.

I would also like to take the opportunity to thank Cray Research for many free hours on their machines during the 1980s and 1990s, IBM for providing me with RISC workstations for private use at home during the 1990s, and Silicon Graphics for their support of my CFD team at George Mason University during the 1990s and to this day. CFD would never have been the same without this support.

Dr David Book undertook the difficult task of reading the first draft of this book, providing many comments and suggestions.

Finally, to those unnamed or unreferenced (and there will always be those): my apologies and thanks. You know who you are.

1 INTRODUCTION AND GENERAL CONSIDERATIONS

Before going into a detailed description of applied Computational Fluid Dynamics (CFD) techniques, it seems proper to define its place among related disciplines. CFD is part of computational mechanics, which in turn is part of simulation techniques. Simulation is used by engineers and physicists to forecast or reconstruct the behaviour of an engineering product or physical situation under assumed or measured boundary conditions (geometry, initial states, loads, etc.). A variety of reasons can be cited for the increased importance that simulation techniques have achieved in recent years.

(a) *The need to forecast performance.* The inability to forecast accurately the performance of a new product can have a devastating effect on companies. The worst nightmare of an aircraft or car manufacturer is to build a prototype which has some hidden flaw that renders it inoperable or seriously degrades market appeal. Of the many examples that could be cited here, we just mention flutter or buzz for aircraft and unforeseen noise or vibrations for cars. With the development costs for new products being so large (about \$4 $\times$ 10^9 for a new aircraft, \$$10^9$ for a new car; these and all subsequent quotations are in US\$ and are accurate in the year 2000), a non-performing product can quickly lead to bankruptcy. The only way to minimize the risk of unexpected performance is through insight, i.e. information. Simulation techniques such as CFD can provide this information.

(b) *Cost of experiments.* Experiments, the only other alternative to simulations, are costly. A day in a large transonic windtunnel costs about \$$10^5$, not counting the personnel costs of planning, preparing the model, analysing the results, etc., as well as the hidden costs of waiting for availability and lost design time. An underground test for a nuclear device costs about \$$10^8$, and for a conventional weapon \$$10^7$. Other large experiments in physics can also command very high prices.

(c) *Impossibility of experiments.* In some instances, experiments are impossible to conduct. Examples are solar and galactic events, atmospheric nuclear explosions (banned after the Test Ban Treaty of 1963), or biomedical situations that would endanger the patient's life.

(d) *Insight.* Most large-scale simulations offer more insight than experiments. A mesh of 2×10^7 gridpoints is equivalent to an experiment with 2×10^7 probes or measuring devices. No experiment that the author is aware of has even nearly this many measuring locations. Moreover, many derived diagnostics (e.g. vorticity, shear, residence time, etc.) can easily be obtained in a simulation, but may be unobtainable in experiments.

(e) *Computer speed and memory.* Computer speed and memory capacity continue to double every 18 months (Moore's law). At the same time, algorithm development continues to

Table 1.1. Increase of problem size

Size	Dimension	Code	Year	Problem	Machine
$>10^2$	2-D	FEFLO20	1983	Airfoil	ICL
$>10^3$	3-D	FEFLO30	1985	Forebody	Cyber-205
$>10^4$	2-D	FEFLO27	1986	Train	Cray-XMP
$>10^5$	3-D	FEFLO72	1989	Train	Cray-2
$>10^6$	3-D	FEFLO74	1991	T-62 Tank	Cray-2
$>10^7$	3-D	FEFLO96	1994	Garage	Cray-M90
$>10^8$	3-D	FEFLO98	1998	Village	SGI Origin 2000

improve accuracy and performance. This implies that ever more realistic simulations can be performed. Table 1.1 summarizes the size of a problem as a function of time from the author's own perspective. Note that in 1983 a problem with more that 1000 finite elements, being run at a university, was considered excessively large!

Although simulations would seem to be more advantageous, the reader should not discount experiments. They provide the only 'reality-check' during the development of new products. However, given the steep decline in computing costs, simulations will certainly reduce the number of required experiments. Boeing estimates indicate that the number of wind-tunnel hours required for the development of the Boeing-747 (1963) was reduced by a factor of 10 for the Boeing-767 (1982) (Rubbert (1988)) and by yet another factor of 10 for the Boeing-777 (1998). Since aerospace is one of the leading fields for simulations, these figures may be indicative of trends to be expected in other manufacturing sectors.

In CFD, the simulation of flows is accomplished by:

(a) solving numerically partial differential equations (PDEs);

(b) following the interaction of a large numbers of particles; or

(c) a combination of both.

The first model is used whenever a continuum assumption for the flow can be made. The second model is used for rarefied flows, where the continuum model is no longer valid. Combinations of fields and particles are used whenever some aspects of a complex problem are best modelled as a continuum, and others by discrete entities, or when the motion of passive marker particles is useful for visualizing flows. Examples where such combinations are commonly employed are plume flows with burning particles and ionized magneto-hydrodynamic flows.

Due to its relevance to the aerospace and defense industries, as well as to most manufacturing processes, CFD has been pursued actively ever since the first digital computers were developed. The Manhattan project was a major testbed and beneficiary of early CFD technology. Concepts such as artificial dissipation date from this time.

CFD, by its very nature, encompasses a variety of disciplines, which have been summarized in Figure 1.1 and may be enumerated in the following order of importance.

(a) *Engineering.* We live in a technology-driven world. Insight for practical engineering purposes is the reason why we pursue CFD. Forget the romantic vision of art for art's sake.

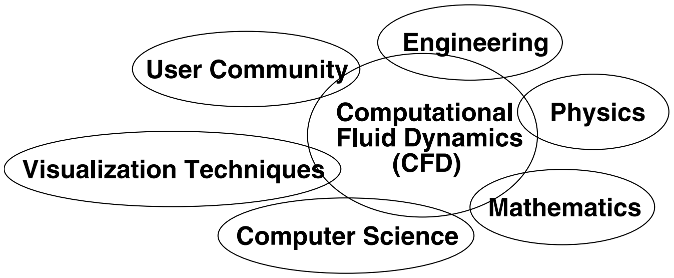


Figure 1.1. The multi-disciplinary nature of CFD

This is engineering, physics, medicine, or any such discipline, and if a CFD code cannot guide the analyst to better products or more understanding, it is simply useless.

(b) *Physics.* Physics explains the phenomena to be simulated for engineering purposes, and provides possible approximations and simplifications to the equations describing the flowfields. For example, the potential approximation, where applicable, represents CPU savings of several orders of magnitude as compared to full Reynolds-averaged Navier–Stokes (RANS) simulations. It is the task of this discipline to outline the domains of validity of the different assumptions and approximations that are possible.

(c) *Mathematics.* Mathematics has three different types of input for CFD applications. These are:

- *classical analysis*, which discusses the nature, boundary conditions, Green kernels, underlying variational principles, adjoint operators, etc., of the PDEs;
- *numerical analysis*, which describes the stability, convergence rates, uniqueness of solutions, well-posedness of numerical schemes, etc.; and
- *discrete mathematics*, which enables the rapid execution of arithmetic operations.

(d) *Computer science.* Computer science has mushroomed into many subdisciplines. The most important ones for CFD are:

- *algorithms*, which describe how to perform certain operations in an optimal way (e.g. the search of items in a list or in space);
- *coding*, so that the final code is portable, easy to modify and/or expand, easy to understand, user-friendly, etc.;
- *software*, which not only encompasses compilers, debuggers and operating systems, but also advanced graphics libraries (e.g. OpenGL); and
- *hardware*, which drives not only the realm of ever-expanding applications that would have been unthinkable a decade ago, but also influences to a large extent the algorithms employed and the way codes are written.

(e) *Visualization techniques*. The vast amounts of data produced by modern simulations need to be displayed in a sensible way. This not only refers to optimal algorithms to filter and traverse the data at hand, but also to ways of seeing this data (plane-cuts, iso-surfaces, X-rays, stereo-vision, etc.).

(f) *User community*. The final product of any CFD effort is a code that is to be used for engineering applications. Successful codes tend to have a user community. This introduces human factors which have to be accounted for: confidence and benchmarking, documentation and education, the individual motivation of the end-users, ego-factors, the not-invented-here syndrome, etc.

1.1. The CFD code

The end-product of any CFD effort is a code that is to be used for engineering applications, or the understanding of physical phenomena that were previously inaccessible. The quality of this tool will depend on the quality of ingredients listed above. Just as a chain is only as strong as its weakest link, a code is only as good as the worst of its ingredients. Given the breadth and variety of disciplines required for a good code, it is not surprising that only a few codes make it to a production environment, although many are written worldwide. Once a CFD code leaves the confines of research, it becomes a *tool*, i.e. a part of the *service industry*. CFD codes, like other tools, can be characterized and compared according to properties considered important by the user community. Some of these are:

- EU: ease of use (problem set-up, user interface, etc.);
- DO: documentation (manuals, help, etc.);
- GF: geometric flexibility;
- TT: turnaround time (set-up to end result);
- BM: benchmarking;
- AC: accuracy;
- SP: speed;
- EX: expandability to new areas/problems.

Like any other product, CFD codes have a customer base. This customer base can be categorized by the number of times a certain application has to be performed. Three main types of end-users may be identified:

(a) those that require a few occasional runs on new configurations to guide them in their designs (e.g. flow simulations in the manufacturing industries and process control);

(b) those that require a large number of runs to optimize highly sophisticated products (e.g. airfoil or wing optimization); and

Table 1.2. Priorities for different user environments

Type of run	No. of runs	Runtime	Desired properties
General purpose/ analysis	$O(1)$	Hours	EU, DO, GF, EX, TT, BM, AC, SP
Design/ optimization	$O(1000)$	Seconds	SP, TT, GF, AC, BM, EU, EX, DO
New physics	$O(10)$	Months	AC, BM, SP, TT, EU, GF, DO, EX

(c) those that require a few very detailed runs on extremely simple geometries in order to understand or discover new physics. These end-users are typically associated with government laboratories. Runs of this kind typically push the limits of tolerance for other users, and their lengths are often the subject of ‘war stories’ (e.g. more than two weeks of continuous CPU time on the fastest machine available).

According to the frequency of runs, the priorities change, as can be seen from Table 1.2.

The message is clear: before designing or comparing codes, one should ask how often the code is to be used on a particular application, how qualified the personnel are, what the maximum allowed turnaround time is, the expected accuracy, and the resources available. Only then can a proper design or choice of codes be made.

1.2. Porting research codes to an industrial context

Going from a research code to an industrial code requires a major change of focus. Industrial codes are characterized by:

- extensive manuals and other documentation;
- a 24-hour hotline answering service;
- a customer support team for special requests/applications;
- incorporation of changes through releases and training.

In short, they require an *organization* to support them. The CFD software and consulting market already exceeds $300 million/year, and is expected to grow rapidly in the coming decade.

At present, CFD is being used extensively in many sectors of the manufacturing industry, and is advancing rapidly into new fields as the more complex physical models become available. In fact, the cost advantages of using CFD have become so apparent to industry that in many areas industry has become the driver, demanding usability, extensions and innovation at a rapid pace. Moreover, large integrators are demanding software standards so that the digital product line extends to their tier 1, 2, 3 suppliers.

1.3. Scope of the book

This book treats the different topics and disciplines required to carry out a CFD run in the order they appear or are required during a run:

(a) data structures (to represent, manage, generate and refine a mesh);

(b) grid generation (to create a mesh);

(c) approximation theory and flow solvers (to solve the PDEs, push particles on the mesh);

(d) interpolation (for particle–mesh solvers, and applications requiring remeshing or externally provided boundary conditions);

(e) adaptive mesh refinement (to minimize CPU and memory requirements); and

(f) efficient use of hardware (to minimize CPU requirements).

This order is different from the historical order in which these topics first appeared in CFD, and the order in which most CFD books are written.

Heavy emphasis is placed on CFD using unstructured (i.e. unordered) grids of triangles and tetrahedra. A number of reasons can be given for this emphasis.

- The only successfully industrialized CFD codes that provide user support, updates and an evolving technology to a large user base are based on unstructured grids. This development parallels the development of finite element codes for computational structural dynamics (CSD) in the 1960s.
- Once the problem has been defined for this more general class of grids, reverting to structured grids is a simple matter.
- A large number of very good books on CFD based on structured (and, to a lesser extent, unstructured) grids exist (e.g. Patankar (1980), Book (1981), Roache (1982), Anderson *et al.* (1984), Oran and Boris (1987), Hirsch (1991), Versteeg and Malalasekera (1996), Hoffmann and Chiang (1998), Ferziger and Peric (1999), Toro (1999), Turek (1999), Gresho and Sani (2000), Wesseling (2001), Blazek (2001), Lomax *et al.* (2001), Donea and Huerta (2002)), and there is no point writing yet another one that repeats most of the material.

As with any technological product, the final result is obtained after seemingly traversing a maze of detours. After all, why use a car (which has to be painted after assembly after mining/producing the iron and all other raw materials . . .) to go to the grocery shop when one can walk the half mile? The answer is that we want to do more with a car than drive half a mile. The same is true for CFD. If the requirement consists of a few simulations of flows past simple geometries, then all this development is not needed. To go the distance to realistic 3-D simulations of flows in or past complex geometries, no other way will do. The reader is therefore asked to be patient. The relevance of some parts will only become apparent in subsequent chapters.

2 DATA STRUCTURES AND ALGORITHMS

Data structures play a major role in any field solver. They enable the rapid access and manipulation of information, allowing significant simplifications in methodologies and code structure, as well as a drastic reduction in CPU requirements. Data structures are of eminent importance for field solvers based on unstructured grids, for which the data is unordered and must be retrieved from lists. It is therefore necessary to devote some attention to this area. This chapter describes the techniques most commonly used to store and manipulate the components of a grid, as well as the relation between the different possible data items and representations. The description starts from the fundamental data items required to describe a grid, proceeding to derived data structures, sorting and searching, and techniques to rapidly scan for spatial proximity. Even though the presentation is done with typical Fortran/C arrays, the principles and ideas are general: anyone handling grids, spatial data or search operations will have to devise similar algorithmic solutions.

2.1. Representation of a grid

If we assume that in order to solve numerically a PDE the geometrical domain is subdivided into small regions called elements, then the most basic task is how to represent in the code the discretization of the problem. For any given subregion or element, its spatial extent must be provided in order to define it. This is done by providing the coordinate information at a sufficient number of discrete points defining the element. For example, in order to define a tetrahedral element in three dimensions, we require the coordinates of the four corner points. Similarly, for a hexahedral element, we require the coordinates of the eight corner points. For elements that assume curved edges or faces, more information needs to be provided. In order to avoid overlapping regions, or voids in the spatial discretization, these points must be shared with all the neighbouring elements. This implies that all the elements surrounding a point should uniquely access the information of its coordinates, as well as other variables (e.g. unknowns). We therefore have two basic sets of data: point data and element data. The relation between the two is given by the so-called connectivity matrix

`inpoel(1:nnode, 1:nelem)`,

where `inpoel`, `nnode` and `nelem` denote, respectively, the connectivity or interdependency matrix between points and element, the number of nodes or points corresponding to one element and the number of elements of the mesh. For element 9 in Figure 2.1 the entries of `inpoel` would be

`inpoel(1,9)=7`, `inpoel(2,9)=8`, `inpoel(3,9)=13`.

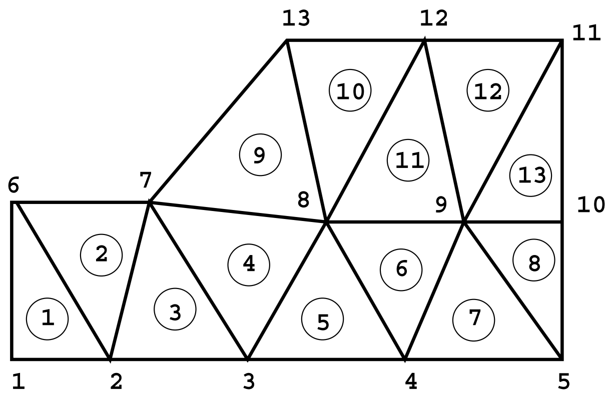


Figure 2.1. Example of a grid

For grids with a mix of element types (e.g. hexahedra, prisms, pyramids and tetrahedra in three dimensions), `nnode` is typically chosen to be the highest possible number of nodes per element. For elements with fewer nodes, the corresponding entries are set to zero. The coordinates are stored in a point array of the form

`coord(1:ndimn, 1:npoin)`,

where `coord`, `ndimn` and `npoin` denote, respectively, the coordinate array, the number of dimensions, and the number of points. The two arrays `inpoel` and `coord` uniquely define the discretization of the geometry. Unknowns may be assumed to be either point arrays or element arrays. In some cases, some variables are stored at the element level, while others are stored at points. For example, several popular finite elements used for incompressible flow simulations store the velocities at points and the pressures in the elements. Unknowns are stored in arrays of the form

`unknp(1:nunkp, 1:npoin)`, `unkne(1:nunke, 1:nelem)`,

where `unknp`, `unkne`, `nunkp` and `nunke` denote, respectively, the arrays of unknowns at points and elements, as well as the number of unknowns at each point and element. For most finite element codes, the arrays defined so far are sufficient to carry out most of the discrete problem set-up and solution. The only exceptions are the boundary conditions: they can either be provided as separate arrays that only require boundary points, e.g.

`bcond(1:nconi, 1:nboup)`,

where `nboup` and `nconi` denote the number of boundary points and the number of boundary condition entries required for each boundary point, respectively, or as separate integer point or element arrays that are flagged appropriately to allow for the proper imposition of boundary conditions. The entry `bcond(1,iboup)` stores the point number; the subsequent entries `bcond(2:nconi,iboup)` contain the information required for the imposition of the desired boundary condition.

2.2. Derived data structures for static data

In many situations that will be described in subsequent chapters, the basic relation between elements and points must be augmented by, or give way to, alternative representations that allow a faster solution of the problem. The following sections describe the most important data structures required, attempting to single out the key technique used to store and retrieve the information required as efficiently as possible, as well as the techniques used to build the data structures themselves. The assumption made in this section is that the primary data (element connectivity, faces, etc.) does not change. Hence these derived data structures are used mainly for static data.

2.2.1. ELEMENTS SURROUNDING POINTS – LINKED LISTS

For the derivation of many of the data structures to follow it is advisable to invert the connectivity matrix `inpoel` that contains the points belonging to an element. This new data structure will allow the rapid access of all the elements surrounding a point. Unlike the number of nodes or points per element, which tends to be constant for most applications, the number of elements surrounding a point can fluctuate widely in a mesh. For example tetrahedral grids tend to vary from one element surrounding a node (a corner element) to over 64. It is clear that making additional room available in a fixed array of the same form as `inpoel` would be extremely wasteful. The most efficient way to store such varying data is through the use of *linked lists*. Instead of one large array, we have two arrays: one for the storage, and the second one to store the starting and ending locations of particular items. For the case of elements surrounding a point, we might use the two arrays:

`esup1(1:mesup)`, `esup2(1:npoin+1)`,

where `esup1` stores the elements, and the ordering is such that the elements surrounding point `ipoin` are stored in locations `esup2(ipoin)+1` to `esup2(ipoin+1)`, as sketched in Figure 2.2.

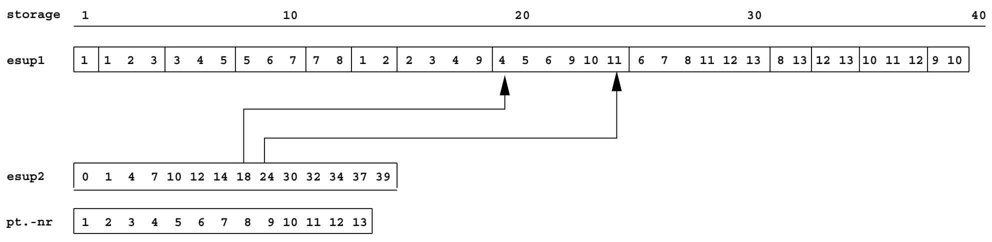


Figure 2.2. Linked list: elements surrounding points

These arrays are constructed in two passes over the elements and two reshuffling passes over the points. In the first pass the storage requirements are counted up. During the second pass the elements surrounding points are stored in `esup1`. The algorithmic implementation is as follows.

Element Pass 1: Count the number of elements connected to each point

```
 Initialize  esup2(1:npoin+1)=0

do ielem=1,nelem                                  ! Loop over the elements
  do inode=1,nnode                          ! Loop over nodes of the element
   ! Update storage counter, storing ahead
     ipoi1=inpoel(inode,ielem)+1
     esup2(ipoi1)=esup2(ipoi1)+1
  enddo
enddo
```

Storage/reshuffling pass 1:

```
do ipoin=2,npoin+1                                  ! Loop over the points
 ! Update storage counter and store
   esup2(ipoin)=esup2(ipoin)+esup2(ipoin-1)
enddo
```

Element pass 2: Store the elements in `esup1`

```
do ielem=1,nelem                                  ! Loop over the elements
  do inode=1,nnode                          ! Loop over nodes of the element
   ! Update storage counter, storing in  esup1
     ipoin=inpoel(inode,ielem)
     istor=esup2(ipoin)+1
     esup2(ipoin)=istor
     esup1(istor)=ielem
  enddo
enddo
```

Storage/reshuffling pass 2:

```
do ipoin=npoin+1,2,-1             ! Loop over points, in reverse order
   esup2(ipoin)=esup2(ipoin-1)
enddo
   esup2( 1)   =0
```

2.2.2. POINTS SURROUNDING POINTS

As with the number of elements that can surround a point, the number of immediate neighbours or points surrounding a point can vary greatly for an unstructured mesh. The best way to store this information is in a linked list. This list will be denoted by

`psup1(1:mpsup), psup2(1:npoin+1)`,

where, as before, `psup1` stores the points, and the ordering is such that the points surrounding point `ipoin` are stored in locations `psup2(ipoin)+1` to `psup2(ipoin+1)`.

The construction of `psup1, psup2` makes use of the element–surrounding-point information stored in `esup1, esup2`. For each point, we can find the elements that surround it from `esup1, esup2`. The points belonging to these elements are the points we are required to store. In order to avoid the repetitive storage of the same point from neighbouring elements, a help-array of the form `lpoin(1:npoin)` is introduced. As will become evident in subsequent sections, help-arrays such as `lpoin` play a fundamental role in the fast construction of derived data structures. As the points are being stored in `psup1` and `psup2`, their entry is also recorded in the array `lpoin`. As new possible nearest-neighbour candidate points emerge from the `esup1`, `esup2` and `inpoel` lists, `lpoin` is consulted to see if they have already been stored.

Algorithmically, `psup1` and `psup2` are constructed as follows:

```
 Initialize:  lpoin(1:npoin)=0
 Initialize:  psup2(1)=0
 Initialize:  istor =0

do ipoin=1,npoin                                  ! Loop over the points
! Loop over the elements surrounding the point
  do iesup=esup2(ipoin)+1,esup2(ipoin+1)
     ielem=esup1(iesup)                              ! Element number
     do inode=1,nnode                    ! Loop over nodes of the element
        jpoin=inpoel(inode,ielem)                      ! Point number
        if(jpoin.ne.ipoin. and .lpoin(jpoin).ne.ipoin) then
        ! Update storage counter, storing ahead, and mark lpoin
          istor=istor+1
          psup1(istor)=jpoin
          lpoin(jpoin)=ipoin
       endif
    enddo
 enddo
  ! Update storage counters
   psup2(ipoin+1)=istor
enddo
```

Alternatives to the use of a help-array such as `lpoin` are an exhaustive (brute-force) search every time a new candidate point appears, or hash tables (Knuth (1973)). It is hard to conceive cases where an exhaustive search would pay off in comparison to the use of help-arrays, except for meshes that have less than 100 points (and in this case, every algorithm will do). Hash tables offer a more attractive alternative. The idea is to introduce an array of the form

`lhash(1:mhash)`,

where `mhash` denotes the maximum possible number of entries or subdivisions of the data range. The key idea is that one can use the sparsity of neighbours of a given point to reduce the storage required from `npoin` locations for `lpoin` to `mhash`, which is typically a fixed, relatively small number (e.g. `mhash=1000`). In the algorithm described above, the array `lhash` replaces `lpoin`. Instead of storing or accessing `lpoin(ipoin)`, one accesses `lhash(1+mod(ipoin,mhash))`. In some instances, neighbouring points will have the same entries in the hash table. These instances are recorded and an exhaustive comparison is carried out in order to see if the same point has been stored more than once.

2.2.3. ELEMENTS SURROUNDING ELEMENTS

A very useful data structure for particle-in-cell (PIC) codes, the tracking of particles for streamline and streakline visualization, and interpolation in general is one that stores the elements that are adjacent to each element across faces (see Figure 2.3). This data structure will be denoted by

`esuel(1:nfael, 1:nelem)`,

where `nfael` is the number of faces per element.

In order to construct this data structure, we must match the points that comprise any of the faces of an element with those of a face of a neighbour element. In order to acquire this information, we again make use of `inpoel`, `esup1` and `esup2`, and help-arrays `lpoin` and `lhelp`.

The `esuel` array is built as follows:

```
Initialize:  lpoin(1:npoin)=0
Initialize:  esuel(1:nfael,1:nelem)=0

do ielem=1,nelem                                     ! Loop over the elements
  do ifael=1,nfael                                ! Loop over the element faces
    ! Obtain the nodes of this face, and store the points in  lhelp
     nnofa=lnofa(ifael)
     lhelp(1:nnofa)=inpoel(lpofa(1:nnofa,ifael),ielem)
     lpoin(lhelp(1:nnofa))=1                              ! Mark in  lpoin
     ipoin=lhelp(1)                                  ! Select a point of the face
    ! Loop over the elements surrounding the point
     do istor=esup2(ipoin)+1,esup2(ipoin+1)
        jelem=esup1(istor)                                ! Element number
        if(jelem.ne.ielem) then
           do jfael=1,nfael                       ! Loop over the element faces
          ! Obtain the nodes of this face, and check
              nnofj=lnofa(jfael)
              if(nnofj.eq.nnofa) then
                 icoun=0
                 do jnofa=1,nnofa               ! Count the nr. of equal points
                    jpoin=inpoel(lpofa(jnofa,jfael),jelem)
                    icoun=icoun+lpoin(jpoin)
                 enddo
                 if(icoun.eq.nnofa) then
                    esuel(ifael,ielem)=jelem ! Store the element
                 endif
              endif
           enddo
        endif
     lpoin(lhelp(1:nnofa))=0                               ! Reset  lpoin
  enddo
enddo
```

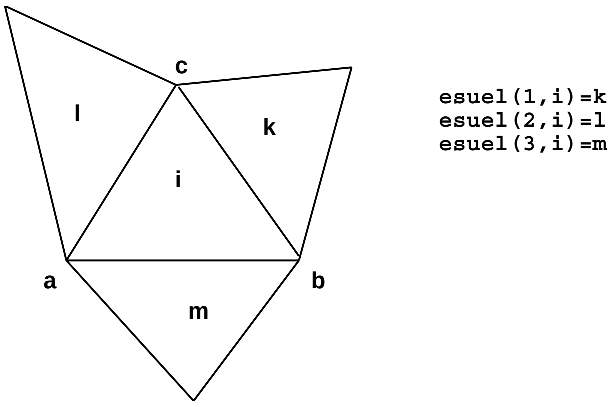


Figure 2.3. `esuel` entries

While `lhelp(1:nnofa)` is a small help-array,

`lnofa(1:nfael)` and `lpofa(1:nnofa,1:nfael)`

contain the data that relates the faces of an element to its respective nodes. Figure 2.4 shows the entries in these arrays for tetrahedral elements.

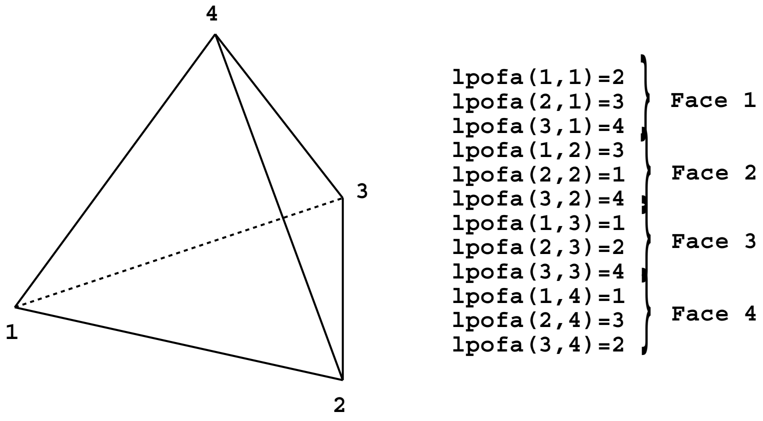


Figure 2.4. Face-data for a tetrahedral element

The generalization to grids with a mix of different elements is straightforward. The algorithm described may be improved in a number of ways. First, the point chosen to access the neighbour elements via `esup1` and `esup2` (`ipoin`) can be chosen to be the one with the smallest number of surrounding elements. Secondly, once the neighbour `jelem` of element `ielem` is found, `ielem` may be stored with very small additional search cost in the corresponding entry of `esuel(1:nfael,jelem)`. This effectively halves CPU requirements. A third improvement, which is relevant mostly to grids with only one type of element and vector machines, is to obtain all the neighbours of the faces coalescing at a point at once. For the case of tetrahedra and hexahedra, this implies obtaining three neighbours for every point visited with `esup1` and `esup2`. The coding of such an algorithm is not trivial, but effective: it again halves CPU requirements on vector machines.

2.2.4. EDGES

Edge representations of discretizations are often used to reduce the CPU and storage requirements of field solvers based on linear (triangular, tetrahedral) elements. They correspond to the graph of the point–surrounding-point combinations. However, unlike `psup1` and `psup2`, they store the endpoints in pairs of the form

$$\texttt{inpoed(1:2,1:nedge)}, \text{ with } \texttt{inpoed}(1, \texttt{iedge}) < \texttt{inpoed}(2, \texttt{iedge}).$$

For linear triangles and tetrahedra the edges correspond exactly to the physical edges of the elements. For bi/trilinear elements, as well as for all higher order elements, this correspondence is lost, as 'internal edges' will appear (see Figure 2.5).

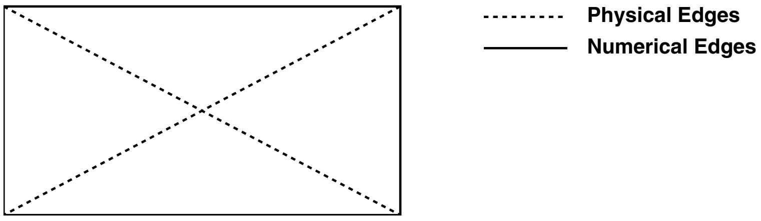


Figure 2.5. Physical and internal edges for quad-elements

The edge data structure can be constructed immediately from `psup1` and `psup2`, or directly, using the same algorithm as described for `psup1` and `psup2` above. The only modifications required are:

(a) an `if` statement in order to satisfy

`inpoed(1,iedge) < inpoed(2,iedge)`, and

(b) changes in notation:

`istor` $\rightarrow$ `nedge`

`psup1` $\rightarrow$ `inpoed`

`psup2` $\rightarrow$ `inpoe1`.

2.2.5. EXTERNAL FACES

External faces are required for a variety of tasks, such as the evaluation of surface normals, surface fluxes, drag and lift integration, etc. One may store the faces in an array of the form

$$\texttt{bface(1:nnofa, 1:nface)},$$

where `nnofa` and `nface` denote the number of nodes per face and the number of faces respectively. If `esuel` is available, the external faces of the grid are readily available: they are simply all the entries for which

`esuel(1:nfael,1:nelem)=0`.

However, a significant storage penalty has to be paid if the external faces are obtained in this way: both `esuel` and `esup1` (required to obtain `esuel` efficiently) are very large arrays. Therefore, alternative algorithms that require far less memory have been devised.

Assuming that the boundary points are known, the following algorithm is among the most efficient.

Step 1: Store faces with all nodes on the boundary

```
Initialize: lpoin(1:npoin)=0
lpoin(bconi(1,1:nconi))=1                 ! Mark all boundary points
Initialize: nface=0
do ielem=1,nelem                          ! Loop over the elements
  do ifael=1,nfael                        ! Loop over the element faces
     ! Obtain the nodes of this face, and store the points in lhelp
     nnofa=lnofa(ifael)
     lhelp(1:nnofa)=inpoel(lpofa(1:nnofa,ifael),ielem)
     ! Count the number of nodes on the boundary
     icoun=0
     do inofa=1,nnofa
        icoun=icoun+lpoin(lhelp(inofa)
     enddo
     if(icoun.eq.nnofa) then
        nface=nface+1                     ! Update face counter
        bface(1:nnofa,nface)=lhelp(1:nnofa))   ! Store the face
    endif
 enddo
enddo
```

After this first step, faces that have all nodes on the boundary, yet are not on the boundary, will be stored twice. An example of the equivalent 2-D situation is illustrated in Figure 2.6.

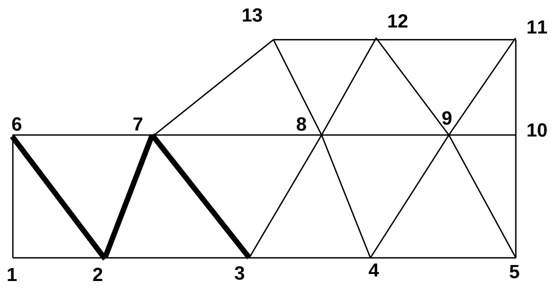


Figure 2.6. Interior faces left after the first pass

These faces are removed in a second step.

Step 2: Remove doubly defined faces. This may be accomplished using a linked list that stores the faces surrounding each point, and then removing the doubly stored faces using a local exhaustive search.

Initialize: `lface(1:nface)=1`
Build the linked list `fsup1(1:nfsup), fsup2(npoin+1)` that stores the faces surrounding each point using the same techniques outlined above for
`esup1(1:nfsup), esup2(npoin+1)`

```
do iboun=1,nboun                          ! Loop over the boundary points
   ipoin=bconi(1,iboun)                                  ! Point number
   ! Outer loop over the faces surrounding the point
   do istor=fsup2(ipoin)+1,fsup2(ipoin+1)-1
      iface=fsup1(istor)                           ! Face number
      if(lface(iface).ne.0) then
          ! See if the face has been marked ⇒
          ! Inner loop over the faces surrounding the point:
         do jstor=istor+1,fsup2(ipoin+1)
            jface=fsup1(jstor)                     ! Face number
            if(iface.ne.jface) then
               if: Points of iface, jface are equal then
                  lface(iface)=0             ! Remove the faces
                  lface(jface)=0
               endif
            endif
         enddo
      endif
   enddo
enddo                                        ! End of loop over the points
```

Retain all faces for which `lface(iface).ne.0`

2.2.6. EDGES OF AN ELEMENT

For the construction of geometrical information of so-called edge-based solvers, as well as for some mesh refinement techniques, the information of which edges belong to an element is necessary. This data structure will be denoted by

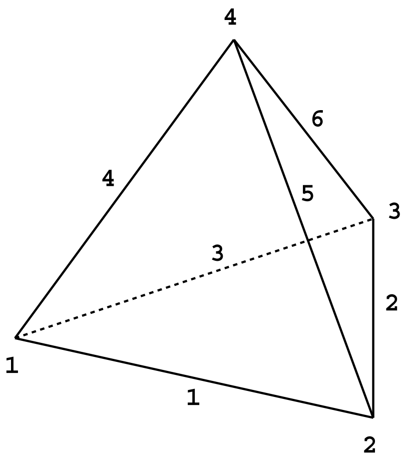
```
inedel(1:nedel,1:nelem)
```

where `nedel` is the number of edges per element.

Given the `inpoel, inpoed` and `inpoe1` arrays, the construction of `inedel` is straightforward:

```
do ielem=1,nelem                                    ! Loop over the elements
   do iedel=1,nedel                               ! Loop over the element edges
      ipoi1=inpoel(lpoed(1,iedel),ielem)
      ipoi2=inpoel(lpoed(2,iedel),ielem)
      ipmin=min(ipoi1,ipoi2)
      ipmax=max(ipoi1,ipoi2)
      ! Loop over the edges emanating from ipmin
      do iedge=inpoe1(ipmin)+1,inpoe1(ipmin+1)
         if(inpoed(2,iedge).eq.ipmax) then
            inedel(iedel,ielem)=iedge
         endif
      enddo
   enddo
enddo
```

The data-array `lpoed(1:2,1:nedel)` contains the nodes corresponding to each of the edges of an element. Figure 2.7 shows the entries for tetrahedral elements. The entries for other elements are straightforward to derive.



```
lpoed(1,1)=1
lpoed(2,1)=2
lpoed(1,2)=2
lpoed(2,2)=3
lpoed(1,3)=3
lpoed(2,3)=1
lpoed(1,4)=1
lpoed(2,4)=4
lpoed(1,5)=2
lpoed(2,5)=4
lpoed(1,6)=3
lpoed(2,6)=4
```

Figure 2.7. Edges of a tetrahedral element

2.3. Derived data structures for dynamic data

In many situations (e.g. during grid generation) the data changes constantly. Should derived data structures be required, then linked lists will not perform satisfactorily. This is because linked lists, in order to achieve optimal storage, require a complete reordering of the data each time a new item is introduced. A better way of dealing with dynamically changing data is the N-tree.

2.3.1. N-TREES

Suppose the following problem is given: find all the faces that surround a given point. One could use a linked list `fsup1, fsup2` as shown above to solve the problem efficiently. Suppose now that the number of faces that surround a point is changing constantly. A situation where this could happen is during grid generation using the advancing front technique. In this case the arrays `fsup1` and `fsup2` have to be reconstructed each time a face is either taken or added to the list. Each reconstruction requires several passes over all the faces and points, making it very expensive. Recall that the main reason for using linked lists was to minimize storage requirements. The alternative is the use of an array of the form `fasup(1:mfsup,1:npoin)`, where `mfsup` denotes the maximum possible number of faces surrounding a point. As this number can be much larger than the average, a great waste of memory is incurred. A compromise between these two extremes can be achieved by using so-called *N-trees*. An array of the form `fasup(1:afsup,1:mfapo)` can be constructed, where `afsup` is a number slightly larger than the average number of faces surrounding a point. The idea is that in most of the instances, locations `1:afsup-1` will be sufficient to store the faces surrounding a point. Should this not be the case, a 'spill-over' contingency is built in by allowing further storage at the bottom of the list. This can be achieved by storing a marker `istor` in location `fasup(afsup,ipoin)` for each point `ipoin`, i.e.

`istor=fasup(afsup,ipoin)`.

As long as there is storage space in locations `1:afsup-1` and `istor.ge.0` a face can be stored. Should more than `afsup-1` faces surround a point, the marker `istor` is set to the negative of the next available storage location at the bottom of the list. An immediate reduction of memory requirements for cases when not all the points are connected to a face (e.g. boundary faces) can be achieved by first defining an array `lpoin(1:npoin)` over the points that contain the place `ifapo` in `fasup` where the storage of the faces surrounding each point starts. It is then possible to reduce `mfapo` to be of the same order as the number of faces `nface`. To summarize this N-tree, we have

`lpoin(ipoin)` : the place `ifapo` in `fasup` where the storage of the faces surrounding point `ipoin` starts,

`fasup( afsup, ifapo) :` > 0 : the number of stored faces
< 0 : the place `jfapo` in `fasup` where the storage of the faces surrounding point `ipoin` is continued,

`fasup(1:afsup-1,ifapo) :` $= 0$: an empty location
> 0 : a face surrounding `ipoin.`

In two dimensions one typically has two faces adjacent to a point, so `afsup=3`, while for 3-D meshes typical values are `afsup=8-10`. Once this N-tree scheme has been set up, storing and/or finding the faces surrounding points is readily done. The process of adding a face to the N-tree `lpoin, fasup` is shown in Figure 2.8. Faces `f1` and `f2` surround point `ipoin`. Face `f3` is also attached to `ipoin`. The storage space in `fasup(1:afsup-1,ifapo)` is already exhausted with `f1`, `f2`. Therefore, the storage of the faces surrounding `ipoin` is continued at the end of the list.

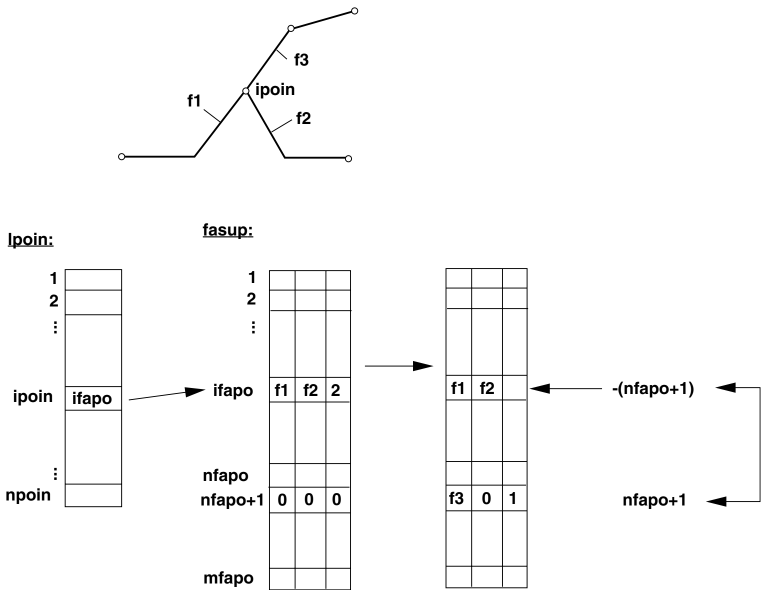


Figure 2.8. Adding an entry to an N-tree

N-trees are not only useful for face/point data. Other applications include edges surrounding points, elements surrounding points, all elements surrounding an element, points surrounding points, etc.

2.4. Sorting and searching

This section will introduce a number of searching and sorting algorithms and their associated data structures. A vast number of searching and sorting techniques exist, and this topic is one of the cornerstones of computer science (Knuth (1973)). The focus here will be on those algorithms that have found widespread use for CFD applications.

Consider the following task: given a set of items (e.g. numbers in an array), order them according to some property or key (e.g. the magnitude of the numbers). Several 'fast sorting' algorithms have been devised (Knuth (1973), Sedgewick (1983)). Among the most often used are: binsort, quicksort and heapsort. While the first two perform very well for static data, the third algorithm is also well suited to dynamic data.

2.4.1. HEAP LISTS

Heap lists are binary tree data structures used commonly in computer science (Williams (1964), Floyd (1964), Knuth (1973), Sedgewick (1983)). The ordering of the tree is accomplished by requiring that the key of any father (root) be smaller than the keys of the two sons (branches). An example of a tree ordered in this manner is given in Figure 2.9, where a possible tree for the letters of the word 'example' is shown. The letters have been arranged according to their place in the alphabet.

A way must now be devised to add or delete entries from such an ordered tree without altering the ordering.

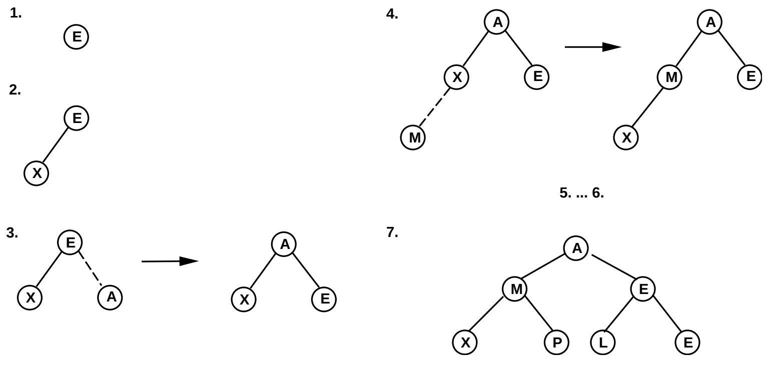


Figure 2.9. Heap list: addition of entries

Suppose the array to be ordered consists of `nelem` elements and is denoted by `lelem(1:nelem)` with associated keys `relem(1:nelem)`. The positions of the son or the father in the heap list `lheap(1:nelem)` are denoted by `ipson` and `ipfath`, respectively. Accordingly, the element numbers of the son or the father in the `lelem` array are denoted by `ieson` and `iefath`. Then `ieson=lheap(ipson)` and `iefath=lheap(ipfath)`. From Figure 2.9 one can see that the two sons of position `ipfath` are located at `ipson1=2∗ipfath` and `ipson2=2*ipfath+1`, respectively.

2.4.1.1. *Adding a new element to the heap list*

The idea is to *add* the new element *at the end* of the tree. If necessary, the internal order of the tree is re-established by comparing father and son pairs. Thus, the tree is traversed from the bottom *upwards*. A possible algorithmic implementation would look like the following:

```
nheap=nheap+1                              ! Increase nheap by one
lheap(nheap)=ienew            ! Place the new element at the end of the heap list
ipson=nheap                 ! Set the positions in the heap list for father and son

while(ipson.ne.1):
   ipfath=ipson/2
   ! Obtain the elements associated with the father and son
   ieson =lheap(ipson)
   iefath=lheap(ipfath)
   if(relem(ieson).lt.relem(iefath)) then
      ! Interchange elements stored in lheap
      lheap(ipson)=iefath
      lheap(ipfath)=ieson
      ipson=ipfath                                 ! Set father as new son
   else
      return                                    ! Key order is now restored
   endif
endwhile
```

In this way, the element with the smallest associated key `relem(ielem)` remains at the top of the list in position `lheap(1)`. The process is illustrated in Figure 2.9, where the letters of the word ‘example’ have been inserted sequentially into the heap list.

2.4.1.2. Removing the element at the top of the heap list

The idea is to *take out* the element at the *top* of the heap list, replacing it by the element at the bottom of the heap list. If necessary, the internal order of the tree is re-established by comparing pairs of father and sons. Thus, the tree is traversed from the top *downwards*.

A possible algorithmic implementation would look like the following:

```
ieout=lheap(1)                              ! Retrieve element at the top
! Place the element stored at the end of the heap list at the top:
lheap(1)=lheap(nheap)
nheap=nheap-1                                           ! Reset nheap
ipfath=1                               ! Set starting position of the father
while(nheap.gt.2*ipfath):
 ! Obtain the positions of the two sons, and the associated elements
   ipson1=2*ipfath
   ipson2=ipson1+1
   ieson1=lheap(ipson1)
   ieson2=lheap(ipson2)
   iefath=lheap(ipfath)
   ! Determine which son needs to be exchanged:
   ipexch=0
   if(relem(ieson1).lt.relem(ieson2)) then
      if(relem(ieson1).lt.relem(iefath)) ipexch=ipson1
   else
      if(relem(ieson2).lt.relem(iefath)) ipexch=ipson2
   endif
   if(ipexch.ne.0) then
      lheap(ipfath)=lheap(ipexch)  ! Exchange father and son entries
      lheap(ipexch)=iefath
      ipfath=ipexch                               ! Reset ipfath
   else
      ! The final position has been reached
      return
   endif
endwhile
```

In this way, the element with the smallest associated key will again remain at the top of the list in position `lheap(1)`. The described process is illustrated in Figure 2.10, where the successive removal of the smallest element (alphabetically) from the previously constructed heap list is shown.

It is easy to prove that both the insertion and the deletion of an element into the heap list will take $O(\log_2(\texttt{nheap}))$ operations (Williams (1964), Floyd (1964)) on average.

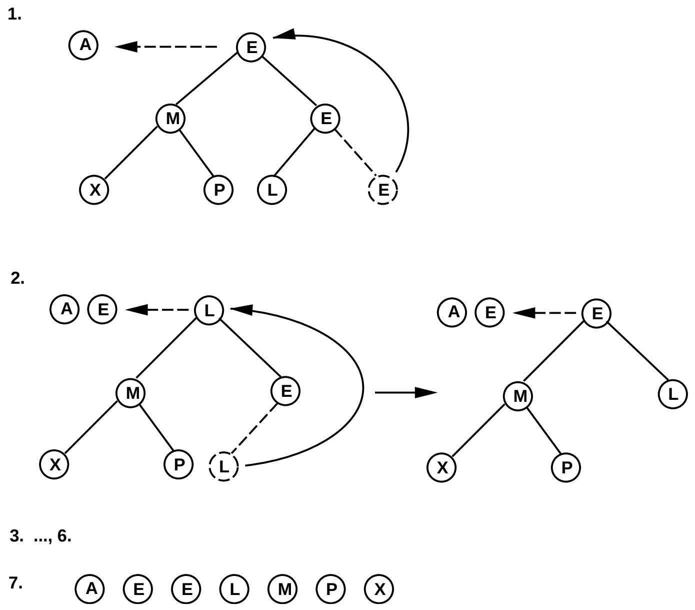


Figure 2.10. Heap list: deletion of entries

2.5. Proximity in space

Given a set of points or a grid, consider the following tasks:

(a) for an arbitrary point, find the gridpoints closest to it;

(b) find all the gridpoints within a certain region of space;

(c) for an arbitrary element of another grid, find the gridpoints that fall into it.

In some instances, the derived data structures discussed before may be applied. For example, case (a) could be solved (for a grid) using the linked list `psup1`, `psup2` described above. However, it may be that more than just the nearest-neighbours are required, making the data structures to be described below more attractive.

2.5.1. BINS

The simplest way to reduce search overheads for spatial proximity is given by bins. The domain in which the data (e.g. points, edges, faces, elements, etc.) falls is subdivided into a regular `nsubx`×`nsuby`×`nsubz` mesh of bricks, as shown in Figure 2.11. These bricks are also called bins. The size of these bins is chosen to be

$$\Delta x = (x_{\max} - x_{\min})/\texttt{nsubx}$$

$$\Delta y = (y_{\max} - y_{\min})/\texttt{nsuby}$$

$$\Delta z = (z_{\max} - z_{\min})/\texttt{nsubz},$$

where x, y, $z_{\text{min / max}}$ denotes the spatial extent of the points considered. The bin into which any point **x** falls can immediately be obtained by evaluating

$$\texttt{isubx} = (x_i - x_{\text{min}})/\Delta x$$

$$\texttt{isuby} = (y_i - y_{\text{min}})/\Delta y$$

$$\texttt{isubz} = (z_i - z_{\text{min}})/\Delta z.$$

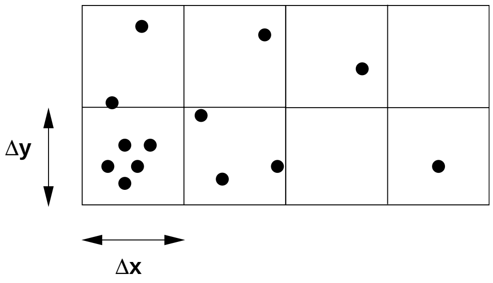


Figure 2.11. Bins

As with all derived data structures, one can either use linked lists, straight storage with maximum allowance or N-trees to store and retrieve the data falling into the bins. If the data is represented as points, one can store point, edge, face or edge centroids in the bins. For items with spatial extent, one typically obtains the *bounding box* (i.e. the box given by the min/max coordinate extent of the edge, face, element, etc.), and stores the item in all the bins it covers.

For the case of point data (coordinates), we might use the two arrays

`lbin1(1:npoin)`, `lbin2(1:nbins+1)`,

where `lbin1` stores the points, and the ordering is such that the points falling into bin `ibin` are stored in locations `lbin2(ibin)+1` to `lbin2(ibin+1)` in array `lbin1` (similar to the linked list sketched in Figure 2.2). These arrays are constructed in two passes over the points and two reshuffling passes over the bins. In the first pass the storage requirements are counted up. During the second pass the points falling into the respective bins are stored in `lbin1`. Assuming the ordering

`ibin = 1 + isubx + nsubx*isuby + nsubx*nsuby*isubz`,

the algorithmic implementation is as follows.

Point pass 0: Obtain bin for each point

```
nsuxy=nsubx*nsuby
do ipoin=1,npoin                              ! Loop over the points
! Obtain bins in each direction and store
   isubx= $(x_i - x_{\text{min}})/\Delta x$
   isuby= $(y_i - y_{\text{min}})/\Delta y$
   isubz= $(z_i - z_{\text{min}})/\Delta z$
   lpbin(ipoin) = 1 + isubx + nsubx*isuby + nsuxy*isubz
enddo
```

Point pass 1: Count number of points falling into each bin

Initialize `lbin2(1:nbins+1)=0`

```
do ipoin=1,npoin                              ! Loop over the points
   ! Update storage counter, storing ahead
   ibin1=lpbin(ipoin)+1
   lbin2(ibin1)=lbin2(ibin1)+1
enddo
```

Storage/reshuffling pass 1:

```
do ibins=2,nbins+1                              ! Loop over the bins
   ! Update storage counter and store
   lbin2(ibins)=lbin2(ibins)+lbin2(ibins-1)
enddo
```

Point pass 2: Store the points in `lbin1`

```
do ipoin=1,npoin                              ! Loop over the points
   ! Update storage counter, storing in lbin1
   ibin =lpbin(ipoin)
   istor=lbin2(ibin)+1
   lbin2(ibin)=istor
   lbin1(istor)=ipoin
enddo
```

Storage/reshuffling pass 2:

```
do ibins=nbins+1,2,-1              ! Loop over bins, in reverse order
   lbin2(ibins)=lbin2(ibins-1)
enddo
   lbin2(1)=0
```

If the data in the vicinity of a location $\mathbf{x}_0$ is required, the bin into which it falls is obtained, and all points in it are retrieved. If no data is found, either the search stops or the region is enlarged (i.e. more bins are consulted) until enough data is found.

For the case of spatial data (bounding boxes of edges, faces, elements, etc.), we can again use the two arrays

`lbin1(1:mstor)`, `lbin2(1:nbins+1)`,

where `lbin1` stores the items (edges, faces, elements, etc.), and the ordering is such that the items falling into bin `ibin` are stored in locations `lbin2(ibin)+1` to `lbin2(ibin+1)` in array `lbin1` (similar to the linked list sketched in Figure 2.2). These arrays are constructed in two passes over the items and two reshuffling passes over the bins. In the first pass the storage requirements are counted up. During the second pass the items covering the respective bins are stored in `lbin1`. Assuming the ordering

```
ibin = 1 + isubx + nsubx*isuby + nsubx*nsuby*isubz,
```

the algorithmic implementation using a help-array for the bounding boxes is below shown for elements.

Element pass 0: Obtain the bin extent for each element

```
nsuxy=nsubx∗nsuby
do ielem=1,nelem                                  ! Loop over the elements
! Obtain the min/max coordinate extent of the element
$\rightarrow$ $x^{el}_{min}$, $x^{el}_{max}$, $y^{el}_{min}$, $y^{el}_{max}$, $z^{el}_{min}$, $z^{el}_{max}$
! Obtain min/max bins in each direction and store
   lebin(1,ielem)= $(x^{el}_{min} - x_{min})/\Delta x$
   lebin(2,ielem)= $(y^{el}_{min} - y_{min})/\Delta y$
   lebin(3,ielem)= $(z^{el}_{min} - z_{min})/\Delta z$
   lebin(4,ielem)= $(x^{el}_{max} - x_{min})/\Delta x$
   lebin(5,ielem)= $(y^{el}_{max} - y_{min})/\Delta y$
   lebin(6,ielem)= $(z^{el}_{max} - z_{min})/\Delta z$
enddo
```

Element pass 1: Count number of bins covered by elements

```
Initialize lbin2(1:nbins+1)=0

do ielem=1,nelem                                  ! Loop over the elements
! Loops over the bins covered by the bounding box
   do isubz=lebin(3,ielem),lebin(6,ielem)
     do isuby=lebin(2,ielem),lebin(5,ielem)
       do isubx=lebin(1,ielem),lebin(4,ielem)
         ibin1 =  2 + isubx + nsubx∗isuby + nsuxy∗isubz
         ! Update storage counter, storing ahead
         lbin2(ibin1)=lbin2(ibin1)+1
       enddo
     enddo
   enddo
enddo
```

Storage/reshuffling pass 1:

```
do ibins=2,nbins+1                                ! Loop over the bins
   ! Update storage counter and store
   lbin2(ibins)=lbin2(ibins)+lbin2(ibins-1)
enddo
```

Element pass 2: Store the elements in `lbin1`

```
do ielem=1,nelem                          ! Loop over the points
! Loops over the bins covered by the bounding box
   do isubz=lebin(3,ielem),lebin(6,ielem)
     do isuby=lebin(2,ielem),lebin(5,ielem)
       do isubx=lebin(1,ielem),lebin(4,ielem)
         ibin = 1 + isubx + nsubx*isuby + nsuxy*isubz
          ! Update storage counter, storing in lbin1
         istor=lbin2(ibin )+1
         lbin2(ibin)=istor
         lbin1(istor)=ielem
       enddo
     enddo
   enddo
enddo
```

Storage/reshuffling pass 2:

```
do ibins=nbins+1,2,-1             ! Loop over bins, in reverse order
   lbin2(ibins)=lbin2(ibins-1)
enddo
   lbin2(1)=0
```

If the data in the vicinity of a location $\mathbf{x}_0$ is required, the bin(s) into which it falls is obtained, and all items in it are retrieved. If no data is found, either the search stops or the region is enlarged (i.e. more bins are consulted) until enough data is found. When storing data with spatial extent, such as bounding boxes, an item may be stored in more than one bin. Therefore, an additional pass has to be performed over the data retrieved from bins, removing data stored repeatedly.

It is clear that bins will perform extremely well for data that is more or less uniformly distributed in space. For unevenly distributed data, better data structures have to be devised. Due to their simplicity and speed, bins have found widespread use for many applications: interpolation for multigrids (Löhner and Morgan (1987), Mavriplis and Jameson (1987)) and overset grids (Meakin and Suhs (1989), Meakin (1993)), contact algorithms in CSD (Whirley and Hallquist (1993)), embedded and immersed grid methods (Löhner *et al.* (2004c)), etc.

2.5.2. BINARY TREES

If the data is unevenly distributed in space, many of the bins will be empty. This will prompt many additional search operations, making them inefficient. A more efficient data structure for such cases is the binary tree. The main concept is illustrated for a series of points in Figure 2.12.

Each point has an associated region of space assigned to it. For each new point being introduced, the tree is traversed until the region of space into which it falls is identified. This region is then subdivided once more by taking the average value of the coordinates of the point already in this region and the new point. The subdivision is taken normal to

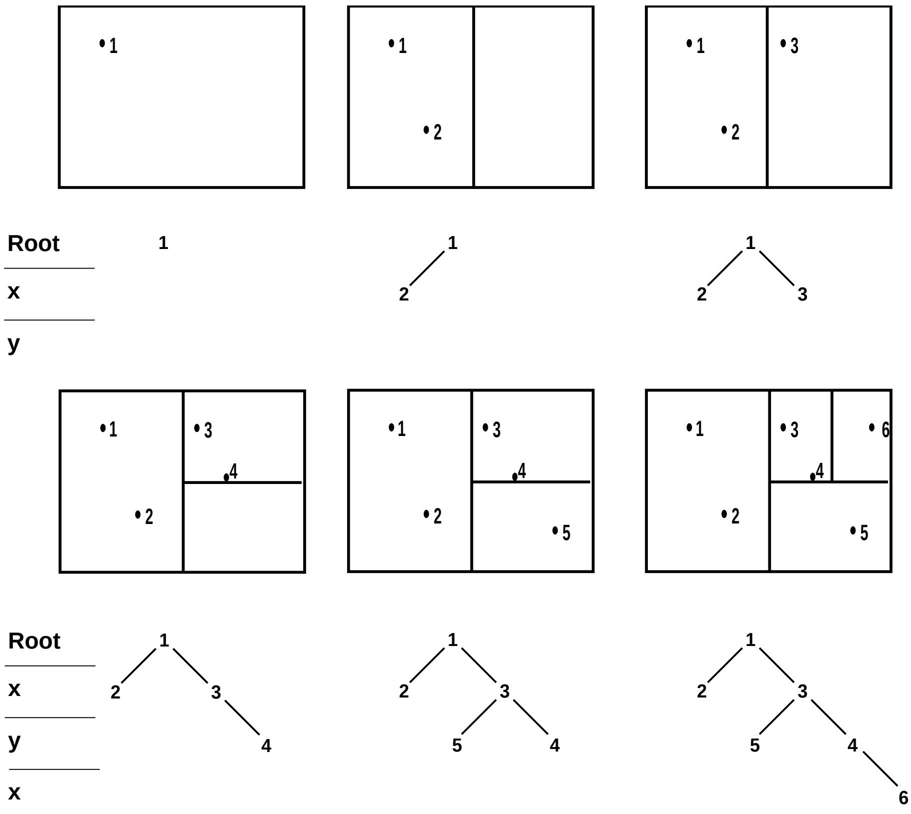


Figure 2.12. Binary tree

the $x/y/z$ directions, and this cycle is repeated as more levels are added to the tree. This alternation of direction has prompted synonyms like coordinate bisection tree (Knuth (1973), Sedgewick (1983)), alternating digital tree (Bonet and Peraire (1991)), etc. When the objective is to identify the points lying in a particular region of space, the tree is traversed from the top downwards. At every instance, all the branches that do not contain the region being searched are excluded. This implies that for evenly distributed points approximately 50% of the database is excluded at every level. The algorithmic complexity of searching for the neighbourhood of any given point is therefore of order $O(N \log_2 N)$. The binary tree is quite efficient when searching for spatial proximity, and is straightforward to code. A shortcoming that is not present in some of the other possible data structures is that the depth of the binary tree, and hence the work required for search, is dependent on the order in which points were introduced. Figure 2.13 illustrates the resulting trees for the same set of six points. Observe that the deepest tree (and hence the highest amount of search work) is associated with the most uniform initial ordering of points, where all points are introduced according to ascending x-coordinate values.

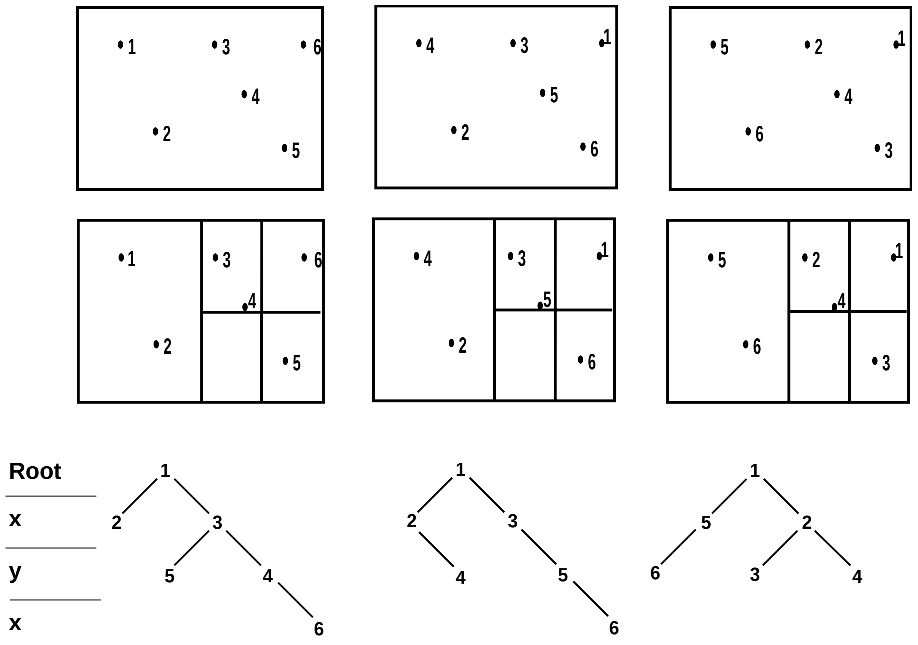


Figure 2.13. Dependence of a binary tree on point introduction order

2.5.3. QUADTREES AND OCTREES

Quadtrees (2-D, subdivision by four) and octrees (3-D, subdivision by eight) are to spatial proximity what N-trees are to linked lists. More than one point is allowed in each portion of space, allowing more data to be discarded at each level as the tree is traversed. The main ideas are described for 2-D regions. The extension to 3-D regions is immediate. Define an array `lquad(1:7,mquad)` to store the points, where `mquad` denotes the maximum number of quads allowed. For each quad `iquad`, store in `lquad(1:7,iquad)` the following information:

`lquad( 7,iquad)` :	<0 :	the quad has been subdivided	
	$=0$:	the quad is empty	
	>0 :	the number of points stored in the quad	
`lquad( 6,iquad)` :	>0 :	the quad the present quad came from	
`lquad( 5,iquad)` :	>0 :	the position in the quad the present quad came from	
`lquad(1:4,iquad)` :	for `lquad(7,iquad)`>0 :	the points stored in this quad	
	for `lquad(7,iquad)`<0 :	the quads into which the present quad was subdivided	

At most four points are stored per quad. If a fifth point falls into the quad, the quad is subdivided into four, and the old points are relocated into their respective quads. Then the fifth point is introduced to the new quad into which it falls. Should the five points end up in the same quad, the subdivision process continues, until a quad with vacant storage space is found. This process is illustrated in Figure 2.14. The newly introduced point E falls into

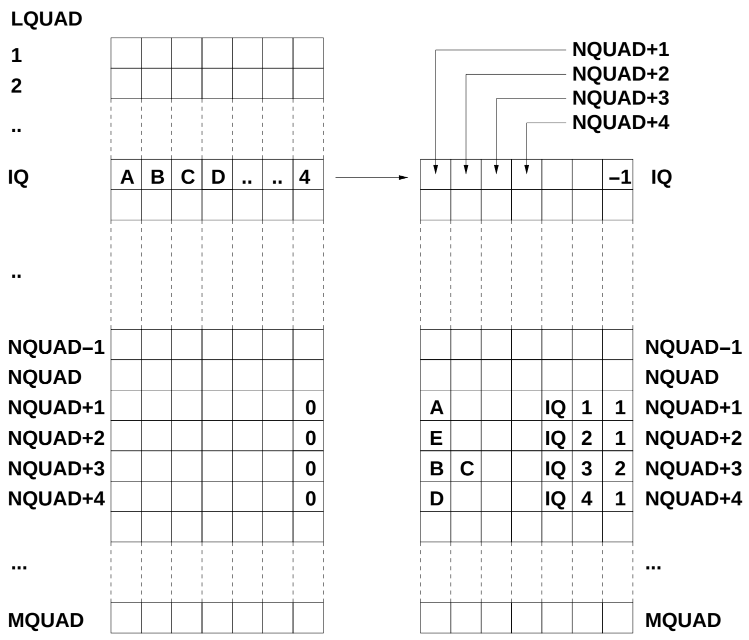


Figure 2.14. Quadtree: addition of entries

the quad `iquad`. As `iquad` already contains the four points A, B, C and D, the quad is subdivided into four. Points A, B, C and D are relocated to the new quads, and point E is added to the new quad `nquad+2`. Figure 2.14 also shows the entries in the `lquad` array, as well as the associated tree structure. In order to find points that lie inside a search region, the quadtree is traversed from the top *downwards*. In this way, those quads that lie outside the search region (and the data contained in them) are eliminated at the highest possible level. For uniformly distributed data, this reduces the data left to be searched by 75% in two dimensions and 87.5% in three dimensions per level. Therefore, for such cases it takes $O(\log_4(N))$ or $O(\log_8(N))$ operations on average to locate all points inside a search region or to find the point closest to a given point. Even though the storage of 2^d items per quad or octant for a d-dimensional problem seems natural, this is by no means a requirement.

In some circumstances, it is advantageous to store more items per quad/octant. The spatial subdivision into 2^d sub-quads/octants remains unchanged, but the number of items n_{so} stored per quad/octant is increased. This will lead to a reduction to $O(\log_{n_{so}}(N))$ ‘jumps’ required on average when trying to locate spatial data. On the other hand, more data will have to be checked once the quadrants/octants covering the spatial domain of interest are found. An example where a larger number of items stored per quad/octant is attractive is given by machines where the access to memory is relatively expensive as compared to operations on local data.

An attractive feature of quadtrees and octrees is that there is a close relationship between the spatial location and the location in the tree. This considerably reduces the number of operations required to find the quads or octants covering a desired search region. Moreover (and notably), the final tree does not depend on the order in which the points were introduced.

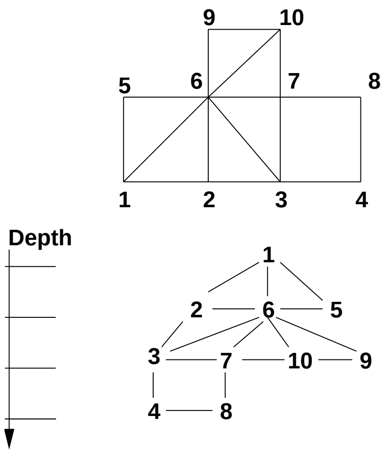


Figure 2.15. Graph of a mesh

The storage of data with spatial extent (given, for example, by bounding boxes) in quad/octrees can lead to difficulties if more then n_{so} items cover the same region of space. In this case, the simple algorithm described above will keep subdividing the quad/octants ad infinitum. A solution to this dilemma is to only allow subdivision of the quad/octants to the approximate size of the bounding boxes of the items to be stored, and then allocate additional 'spill over' storage at the end of the list.

2.6. Nearest-neighbours and graphs

As we saw from the previous sections, a mesh is defined by two lists: one for the elements and one for the points. An interesting notion that requires some definition is what constitutes the neighbourhood of a point. The nearest-neighbours of a point are all those points that are part of the elements surrounding it. With this definition, a graph can be constructed, starting from any given point, and moving out in layers as shown in Figure 2.15 for a simple mesh. In order to construct the graph, use is made of the linked list `psup1, psup2` described above. Depending on the starting position, the number of layers required to cover all points will vary. The maximum number of layers required to cover all points in this way is called the *graph depth* of the mesh, and it plays an important role in estimating the efficiency of iterative solvers.

2.7. Distance to surface

The need to find (repeatedly) the distance to a domain surface or an internal surface is common to many CFD applications. Some turbulence models used in RANS solvers require the distance to walls in order to estimate the turbulent viscosity (Baldwin and Lomax (1978)). For moving mesh applications, the distance to moving walls may be used to 'rigidize' the motion of elements in near-wall regions (Löhner and Yang (1996)). For multiphase problems the distance to the liquid/gas interface is necessary to estimate surface tension (Sethian (1999)).

Let us consider the distance to walls first. A brute force calculation of the shortest distance from any given point to the wall faces would require $O(N_p \cdot N_b)$, where N_p denotes the number of points and N_b the number of wall boundary points. This is clearly unacceptable for 3-D problems, where $N_b \approx N_p^{2/3}$. A better way to construct the desired distance function requires a combination of the point–surrounding point linked list `psup1, psup2`, a heap list for the points, the faces on the boundary `bface`, as well as the faces-surrounding points linked list `fsup1, fsup2`. The algorithm consists of two parts, which may be summarized as follows (see Figure 2.16).

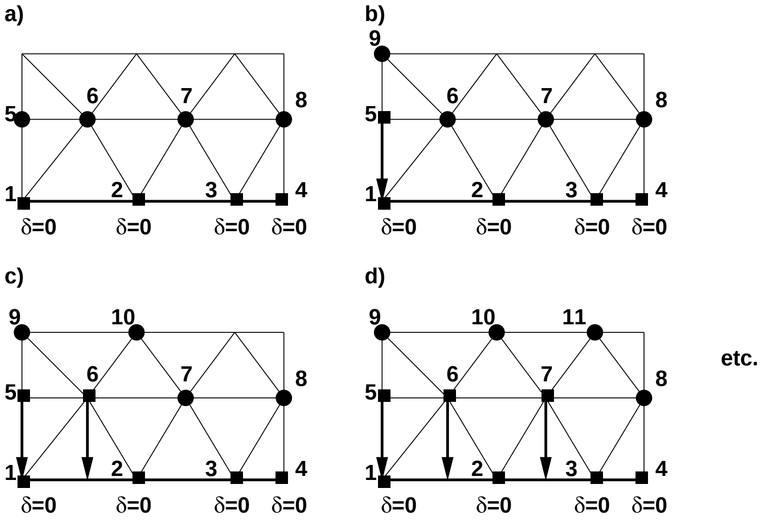


Figure 2.16. Distance to wall

Part 1: Initialization

Obtain the faces on the surfaces from which distance is to be measured:
`bface(1:nnofa,1:nface)`
Obtain the list of faces surrounding faces for `bface`
$\rightarrow$ `fsufa(1:nsifa,1:nface)`
Set all points as unmarked: `lhofa(1:npoin)=0`

```
do iface=1,nface                          ! Loop over the boundary faces
   do inofa=1,nnofa                       ! Loop over the nodes of the face
      ipoin=bface(inofa,iface)                          ! Point number
      if(lhofa(ipoin).eq.0) then          ! The point has not been marked
         lhofa(ipoin)=iface                 ! Set current face as host face
         diswa(ipoin)=0.0
         Introduce ipoin into the heap list with key diswa(ipoin)
         (→ updates nheap)
      endif
   enddo
enddo
```

Part 2: Domain points

```
while(nheap.gt.0):
Retrieve the point ipmin with smallest distance to wall
(this is the point at the top of the heap list)
! Find the points surrounding ipmin
  ipsp0=psup2(ipmin)+1                     ! Storage locations in psup1
  ipsp1=psup2(ipmin+1)
  do ipsup=ipsp0,ipsp1         ! Loop over the points surrounding ipmin
     ipoin=psup1(ipsup)
     if(lhofa(ipoin).le.0) then             ! The wall distance is unknown
        ifsta=lhofa(ipmin)                        ! Starting face for search
        Starting with ifsta, find the shortest distance to the wall
        $\rightarrow$ ifend, disfa
        lhofa(ipoin)=ifend                   ! Set current face as host face
        diswa(ipoin)=disfa                                   ! Set distance
        ! Find the points surrounding ipoin
        jpsp0=psup2(ipoin)+1
        jpsp1=psup2(ipoin+1)
        do jpsup=jpsp0,jpsp1
           jpoin=psup1(jpsup)
           if(lhofa(jpoin).eq.0) then
              The wall-distance is unknown $\Rightarrow$
              Introduce jpoin into the heap list with key disfa
              ($\rightarrow$ updates nheap)
              lhofa(jpoin)=-1             ! Mark as in heap/ unknown dista
           endif
        enddo
     endif
    enddo
endwhile
```

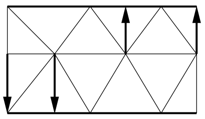


Figure 2.17. Centre line uncertainty

The described algorithm will not work well if we only have one point between walls, as shown in Figure 2.17. In this case, depending on the neighbour closest to a wall, the ‘wrong’ wall may be chosen. In general, the distance to walls for the centre points will have an uncertainty of one gridpoint. Except for these details, the algorithm described will yield the correct distance to walls with an algorithmic complexity of $O(N_p)$, which is clearly much superior to a brute force approach.

For the case of internal surfaces the algorithm described above requires some changes in the initialization. Assuming, without loss of generality, that we have a function $\Phi(\mathbf{x})$ such that the internal surface is defined by $\Phi(\mathbf{x}) = 0$, one may proceed as follows.

Part 1: Initialization

```
Set all points as unmarked: lpoin(1:npoin)=0
Set all points as unmarked: diswa(1:npoin)=1.0d+16

do ielem=1,nelem                                   ! Loop over the elements
If the element contains: $\Phi(\mathbf{x}) = 0$:
$\rightarrow$ Obtain the face(s), storing in bface(1:nnofa,1:nface)
  do inode=1,nnode                     ! Loop over the nodes of the element
  ipoin=intmat(inode,ielem)                                        ! Point
  Obtain the distance of the face(s) to ipoin $\rightarrow$ dispo
  if(dispo.lt.diswa(ipoin) then
     lpoin(ipoin)=iface
     diswa(ipoin)=dispo
  endif
enddo

Obtain the list of faces surrounding faces for bface
$\rightarrow$ fsufa(1:nsifa,1:nface)

Set all points as unmarked: lhofa(1:npoin)=0

do ipoin=1,npoin                                     ! Loop over the points
   if(lpoin(ipoin).gt.0) then                  ! Point is close to surface
      lhofa(ipoin)=lpoin(ipoin)          ! Set current face as host face
      Introduce ipoin into the heap list with key diswa(ipoin)
      ($\rightarrow$ updates nheap)
   endif
enddo
```

From this point onwards, the algorithm is the same as the one outlined above.

3 GRID GENERATION

Numerical methods based on the spatial subdivision of a domain into polyhedra or elements immediately imply the need to generate a mesh. With the availability of more versatile field solvers and powerful computers, analysts attempted the simulation of ever increasing geometrical and physical complexity. At some point (probably around 1985), the main bottleneck in the analysis process became the grid generation itself. The late 1980s and 1990s have seen a considerable amount of effort devoted to automatic grid generation, as evidenced by the many books (e.g. Carey (1993, 1997), George (1991a), George and Borouchaki (1998), Frey (2000)) and conferences devoted to the subject (e.g. the bi-annual International Conference on Numerical Grid Generation in Computational Fluid Dynamics and Related Fields (Sengupta *et al.* (1988), Arcilla *et al.* (1991), Weatherill *et al.* (1993b))) and the yearly Meshing Roundtable organized by Sandia Laboratories (1992–present) resulting in a number of powerful and, by now, mature techniques.

Mesh types are as varied as the numerical methodologies they support, and can be classified according to:

- conformality;
- surface or body alignment;
- topology; and
- element type.

The different mesh types have been sketched in Figure 3.1.

Conformality denotes the continuity of neighbouring elements across edges or faces. Conformal meshes are characterized by a perfect match of edges and faces between neighbouring elements. Non-conforming meshes exhibit edges and faces that do not match perfectly between neighbouring elements, giving rise to so-called hanging nodes or overlapped zones.

Surface or body alignment is achieved in those meshes whose boundary faces match the surface of the domain to be gridded perfectly. If faces are crossed by the surface, the mesh is denoted as being non-aligned.

Mesh topology denotes the structure or order of the elements. The three possibilities here are:

1. micro-structured, i.e. each point has the same number of neighbours, implying that the grid can be stored as a logical `i,j,k` assembly of bricks;
2. micro-unstructured, i.e. each point can have an arbitrary number of neighbours; and
3. macro-unstructured, micro-structured, where the mesh is assembled from groups of micro-structured subgrids.

Applied Computational Fluid Dynamics Techniques: An Introduction Based on Finite Element Methods, Second Edition.
Rainald Löhner © 2008 John Wiley & Sons, Ltd. ISBN: 978-0-470-51907-3

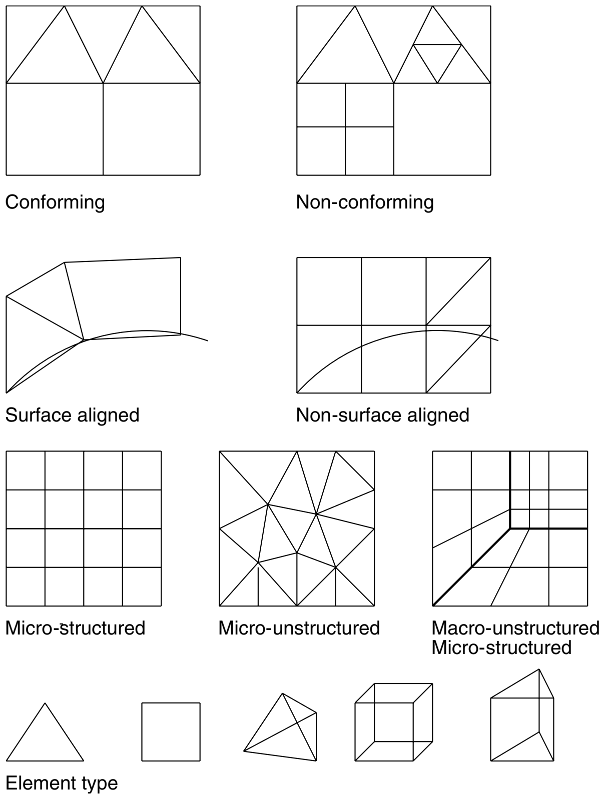


Figure 3.1. Characterization of different mesh types

Element type describes the polyhedron used to discretize space. Typical element types include triangles and quads for 2-D domains, and tetrahedra, prisms and bricks for 3-D domains.

In principle, any of the four classifications can be combined randomly, resulting in a very large number of possible grid types. However, most of these combinations prove worthless. As an example, consider an unstructured grid of tetrahedra that is not body conforming. There may be some cases where this grid optimally solves a problem, but in most cases Cartesian hexahedral cells will lead to a much faster solver and a better suited solution. At present, the main contenders for generality and ease of use in CFD are as follows.

(a) *Multiblock grids*. These are conformal, surface-aligned, macro-unstructured, micro structured grids consisting of quads or bricks. The boundaries between the individual micro-structured grids can either be conforming or non-conforming. The latter class of multiblock grids includes the possibility of overlapped micro-structured grids, also known as Chimera grids (Benek *et al.* (1985), Meakin and Suhs (1989), Dougherty and Kuan (1989)).

(b) *Adaptive Cartesian grids*. These are non-conformal, non-surface-aligned, micro-unstructured grids consisting of quads or bricks. The geometry is simply placed into an initial coarse Cartesian grid that is refined further until a proper resolution of the geometry is achieved (Melton *et al.* (1993), Aftosmis *et al.* (2000)). The imposition of proper boundary conditions at the edges or faces that intersect the boundary is left to the field solver.

(c) *Unstructured uni-element grids*. These are conformal, surface-aligned, micro-unstructured grids consisting of triangles or tetrahedra.

Consider the task of generating an arbitrary mesh in a given computational domain. The information required to perform this task is:

(a) a description of the bounding surfaces of the domain to be discretized;

(b) a description of the desired element size, shape and orientation in space;

(c) the choice of element type; and

(d) the choice of a suitable method to achieve the generation of the desired mesh.

Historically, the work progressed in the opposite order to the list given above. This is not surprising, as the same happened when solvers were being developed. In the same way that the need for grid generation only emerged after field solvers were sufficiently efficient and versatile, surface definition and the specification of element size and shape only became issues once sufficiently versatile grid generators were available.

3.1. Description of the domain to be gridded

There are two possible ways of describing the surface of a computational domain:

(a) using analytical functions; and

(b) via discrete data.

3.1.1. ANALYTICAL FUNCTIONS

This is the preferred choice if a CAD-CAM database exists for the description of the domain, and has been used almost exclusively to date. Splines, B-splines, non-uniform rational B-splines (NURBS) surfaces (Farin (1990)) or other types of functions are used to define the surface of the domain. An important characteristic of this approach is that the surface is continuous, i.e. there are no ‘holes’ in the information. While generating elements on the surface, the desired element size and shape is taken into consideration via mappings (Löhner and Parikh (1988b), Lo (1988), Peiro *et al.* (1989), Nakahashi and Sharov (1995), Woan (1995)).

3.1.2. DISCRETE DATA

Here, instead of functions, a cloud of points or an already existing surface triangulation describes the surface of the computational domain. This choice may be attractive when no CAD-CAM database exists. Examples are remote sensing data, medical imaging data, data sets from virtual reality (training, simulation, film making) and data sets from computer games. For ‘reverse engineering’ or ‘clay to CAD’, commercial digitizers can gather surface point information at speeds higher than 30 000/points/s, allowing a very accurate description of a scaled model or the full configuration (Merriam and Barth (1991b)). The surface definition is completed with the triangulation of the cloud of points. This triangulation process is far from trivial and has been the subject of major research and development efforts (see, e.g., Choi *et al.* (1988), Hoppe *et al.* (1992, 1993)). This approach can lead to a discontinuous surface description. In order not to make any mistakes when discretizing the surface during mesh generation, only the points given in the cloud of points should be selected. The re-triangulation required for surfaces defined in this way has been treated by Hoppe *et al.* (1993), Lo (1995), Löhner (1996), Cebral and Löhner (1999, 2001), Frey and Borouchaki (1998), Frey and George (2000), Ito and Nakahashi (2002), Tilch and Löhner (2002), Surazhsky and Gotsman (2003) and Wang *et al.* (2007).

The current incompatibility of formats for the description of surfaces makes the surface definition by far the most labour-intensive task of the CFD analysis process. Grid generation, flow solvers and visualization are processes that have been automated to a high degree. This is not the case with surface definition, and may continue to be so for a long time. It may also have to do with the nature of analysis: for a CFD run a vast portion of CAD-CAM data has to be filtered out (nuts, bolts, etc., are not required for the surface definition required for a standard CFD run), and the surface patches used by the designers seldomly match, leaving gaps or overlap regions that have to be treated manually. For some recent work on ‘geometry cleaning’ see Dawes (2005, 2006).

3.2. Variation of element size and shape

After the surface of the domain to be gridded has been described, the next task is to define how the element size and shape should vary in space. The parameters required to generate an arbitrary element are shown in Figure 3.2.

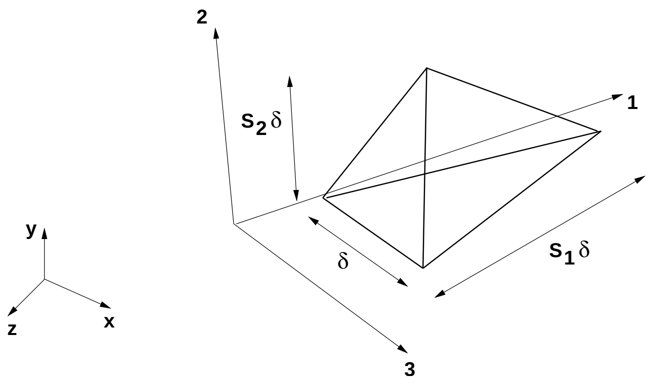


Figure 3.2. Parameters required for an arbitrary element

They consist of the side distance parameter δ, commonly known as the element length, two stretching parameters S_1, S_2, and two associated stretching directions $\mathbf{s}_1$, $\mathbf{s}_2$. Furthermore, the assumption that $S_3 = 1$, $\mathbf{s}_3 = \mathbf{s}_1 \times \mathbf{s}_2$ is made.

3.2.1. INTERNAL MEASURES OF GRID QUALITY

The idea here is to start from a given surface mesh. After the introduction of a new point or element, the quality of the current grid or front is assessed. Then, a new point or element is introduced in the most critical region. This process is repeated until a mesh that satisfies a preset measure of quality is achieved (Holmes and Snyder (1988), Huet (1990)). This technique works well for equilateral elements, requiring minimal user input. On the other hand, it is not very general, as the surface mesh needs to be provided as part of the procedure.

3.2.2. ANALYTICAL FUNCTIONS

In this case, the user codes in a small subroutine the desired variation of element size, shape and orientation in space. Needless to say, this is the least general of all procedures, requiring new coding for every new problem. On the other hand, if the same problem needs to be discretized many times, an optimal discretization may be coded in this way. Although it may seem inappropriate to pursue such an approach within unstructured grids, the reader may be reminded that most current airfoil calculations are carried out using this approach.

3.2.3. BOXES

If all that is required are regions with uniform mesh sizes, one may define a series of boxes in which the element size is constant. For each location in space, the element size taken is the smallest of all the boxes containing the current location. When used in conjunction with surface definition via quad/octrees or embedded Cartesian grids, one can automate the point distribution process completely in a very elegant way (Yerry and Shepard (1984), Shephard and Georges (1991), Aftosmis *et al.* (2000), Dawes (2006, 2007)).

3.2.4. POINT/LINE/SURFACE SOURCES

A more flexible way that combines the smoothness of functions with the generality of boxes or other discrete elements is to define sources. The element size for an arbitrary location $\mathbf{x}$ in space is given as a function of the closest distance to the source, $r(\mathbf{x})$. Consider first the line source given by the points $\mathbf{x}_1$, $\mathbf{x}_2$ shown in Figure 3.3.

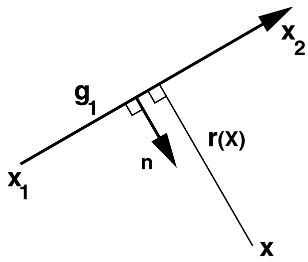


Figure 3.3. Line source

The vector $\mathbf{x}$ can be decomposed into a portion lying along the line and the normal to it. With the notation of Figure 3.3, we have

$$\mathbf{x} = \mathbf{x}_1 + \xi\mathbf{g}_1 + \alpha\mathbf{n}. \tag{3.1}$$

The ξ can be obtained by scalar multiplication with $\mathbf{g}_1$ and is given by

$$\xi = \frac{(\mathbf{x} - \mathbf{x}_1) \cdot \mathbf{g}_1}{\mathbf{g}_1 \cdot \mathbf{g}_1}. \tag{3.2}$$

By forcing the value of ξ to be on the line $\mathbf{x}_1 : \mathbf{x}_2$,

$$\xi' = \max(0, \min(1, \xi)), \tag{3.3}$$

the distance between the point $\mathbf{x}$ and the closest point on the line source is given by

$$\delta(\mathbf{x}) = |\mathbf{x}_1 + \xi'\mathbf{g}_1 - \mathbf{x}|. \tag{3.4}$$

Consider next the surface source element given by the points $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3$ as shown in Figure 3.4.

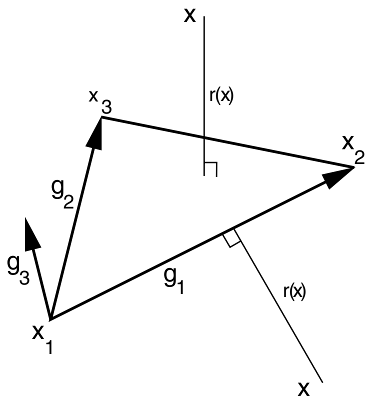


Figure 3.4. Surface source

The vector $\mathbf{x}$ can be decomposed into a portion lying in the plane given by the surface source points and the normal to it. With the notation of Figure 3.4, we have

$$\mathbf{x} = \mathbf{x}_1 + \xi\mathbf{g}_1 + \eta\mathbf{g}_2 + \gamma\mathbf{g}_3, \tag{3.5}$$

where

$$\mathbf{g}_3 = \frac{\mathbf{g}_1 \times \mathbf{g}_2}{|\mathbf{g}_1 \times \mathbf{g}_2|}. \tag{3.6}$$

By using the contravariant vectors $\mathbf{g}^1, \mathbf{g}^2$, where $\mathbf{g}_i \cdot \mathbf{g}^j = \delta_i^j$, we have

$$\xi = (\mathbf{x} - \mathbf{x}_1) \cdot \mathbf{g}^1, \quad \eta = (\mathbf{x} - \mathbf{x}_1) \cdot \mathbf{g}^2, \quad \zeta = 1 - \xi - \eta. \tag{3.7}$$

Whether the point $\mathbf{x}$ lies 'on the surface' can be determined by the condition

$$0 \le \xi, \eta, \zeta \le 1. \tag{3.8}$$

If this condition is violated, the point $\mathbf{x}$ is closest to one of the given edges, and the distance to the surface is evaluated by checking the equivalent line sources associated with the edges.

If (3.8) is satisfied, the closest distance between the surface and the point is given by

$$\delta(\mathbf{x}) = |(1 - \xi - \eta)\mathbf{x}_1 + \xi\mathbf{x}_2 + \eta\mathbf{x}_3 - \mathbf{x}|. \tag{3.9}$$

As one can see, the number of operations required to determine $\delta(\mathbf{x})$ is not considerable if one can pre-compute and store the geometrical parameters of the sources ($\mathbf{g}_i$, $\mathbf{g}^i$, etc.). A line source may be obtained by collapsing two of the three points (e.g. $\mathbf{x}_3 \to \mathbf{x}_2$), and a point source by collapsing all three points into one. In order to reduce the internal complexity of a code, it is advisable to only work with one type of source. Given that the most general source is the surface source, line and point sources are prescribed as surface sources, leaving a small distance between the points to avoid numerical problems (e.g. divisions by zero).

Having defined the distance from the source, the next step is to select a function that is general yet has a minimum of input to define the element size as a function of distance. Typically, a small element size is desired close to the source and a large element size away from it. Moreover, the element size should be constant (and small) in the vicinity $r < r_0$ of the source. An elegant way to satisfy these requirements is to work with functions of the transformed variable

$$\rho = \max\left(0, \frac{r(\mathbf{x}) - r_0}{r_1}\right). \tag{3.10}$$

For obvious reasons, the parameter r_1 is called the scaling length. Commonly used functions of ρ to define the element size in space are:

(a) *power laws*, given by expressions of the form (Löhner *et al.* (1992))

$$\delta(\mathbf{x}) = \delta_0[1 + \rho^\gamma], \tag{3.11}$$

with the four input parameters δ_0, r_0, r_1, γ; where, typically, $1.0 \le \gamma \le 2.0$;

(b) *exponential functions*, which are of the form (Weatherill (1992), Weatherill and Hassan (1994))

$$\delta(\mathbf{x}) = \delta_0 e^{\gamma\rho}, \tag{3.12}$$

with the four parameters δ_0, r_0, r_1, γ;

(c) *polynomial expressions*, which avoid the high cost of exponents and logarithms by employing expressions of the form

$$\delta(\mathbf{x}) = \delta_0\left[1 + \sum_{i=1}^{n} a_i\rho^i\right], \tag{3.13}$$

with the $n + 3$ parameters δ_0, r_0, r_1, a_i, where, typically, quadratic polynomials are employed (i.e. $n = 2$, implying five free parameters).

Given a set of m sources, the minimum is taken whenever an element is to be generated:

$$\delta(\mathbf{x}) = \min(\delta_1, \delta_1, \ldots, \delta_m). \tag{3.14}$$

Some authors, notably Prizadeh (1993a), have employed a smoothing procedure to combine background grids with sources. The effect of this smoothing is a more gradual increase in element size away from regions where small elements are required.

Sources offer a convenient and general way to define the desired element size in space. They may be introduced rapidly in interactive mode with a mouse-driven menu once the surface data is available. For moving or tumbling bodies the points defining the sources relevant to them may move in synchronization with the corresponding body. This allows high-quality remeshing when required for this class of problem (Löhner (1990b), Baum and Löhner (1993), Baum *et al.* (1995, 1996), Löhner *et al.* (1999), Tremel *et al.* (2006)). On the other hand, sources suffer from one major disadvantage: at every instance, the generation parameters of *all* sources need to be evaluated. For a distance distribution given by equations (3.10)–(3.14), it is very difficult to 'localize' the sources in space in order to filter out the relevant ones. As an example, consider the 1-D situation shown in Figure 3.5. Although sources S_3, S_5 are closer to the shaded region than any other source, the source that yields the smallest element size δ in this region is S_2.

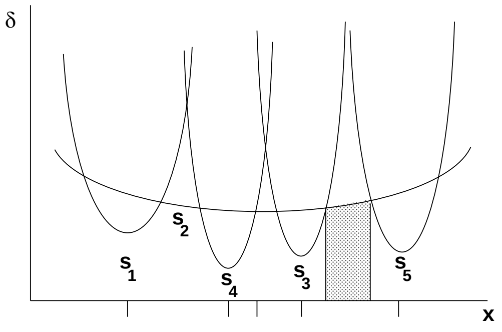


Figure 3.5. Minimum element size from different surface sources

The evaluation of the minimum distance obtained over the sources may be vectorized in a straightforward way. Nevertheless, a large number of sources ($N_s > 100$) will have a marked impact on CPU times, even on a vector machine. Experience shows that the large number of sources dictated by some complex geometries can lead to situations where the dominant CPU cost is given by the element-size evaluations of the sources, not the grid generation method itself.

3.2.5. BACKGROUND GRIDS

Here, a coarse grid is provided by the user. At each of the nodes of this background grid, the element size, stretching and stretching direction are specified. Whenever a new element or point is introduced during grid generation, this background grid is interrogated to determine the desired size and shape of elements. While very general and flexible, the input of suitable background grids for complex 3-D configurations can become tedious. The main use of background grids is for adaptive remeshing: given a first grid and a solution, the element size and shape for a mesh that is more suited for the problem at hand can be determined from an error indicator. With this information, a new grid can be generated by taking the current grid as the background grid (Peraire *et al.* (1987, 1988), Löhner (1988b, 1990b), Peraire *et al.* (1990, 1992b)). For this reason, background grids are still prevalent in most unstructured grid generators, and are employed in conjunction with sources or other means of defining element size and shape.

3.2.6. ELEMENT SIZE ATTACHED TO CAD DATA

For problems that require gridding complex geometries, the specification of proper element sizes can be a tedious process. Conventional background grids would involve many tetrahedra, whose generation is a labour-intensive, error-prone task. Point, line or surface sources are not always appropriate either. Curved 'ridges' between surface patches, as sketched in Figure 3.6, may require many line sources.

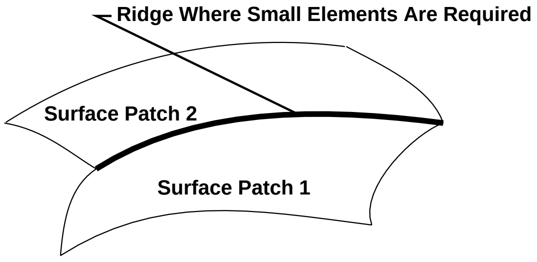


Figure 3.6. Specifying small element size for curved ridges

Similarly, the specification of gridding parameters for surfaces with high curvature may require many surface sources. The net effect is that for complex geometries one is faced with excessive labour costs (background grids, many sources) and/or CPU requirements during mesh generation (many sources).

A better way to address these problems is to attach element size (or other gridding parameters) directly to CAD data. For many problems, the smallest elements are required close to the boundary. Therefore, if the element size for the points of the current front is stored, the next element size may be obtained by multiplying it with a user-specified increase factor c_{incr}. The element size for each new point introduced is then taken as the minimum obtained from the background grid δ_{bg}, the sources δ_s and the minimum of the point sizes corresponding to the face being deleted, multiplied by a user-specified increase factor c_i:

$$\delta = \min(\delta_{bg}, \delta_s, c_i \min(\delta_A, \delta_B, \delta_C)). \tag{3.15}$$

Typical values for c_i are $c_i = 1.2$–1.5.

3.2.7. ADAPTIVE BACKGROUND GRIDS

As was seen from the previous sections, the specification of proper element size and shape in space can be a tedious, labour-intensive task. Adaptive background grid refinement may be employed in order to reduce the amount of user intervention to a minimum. As with any other mesh refinement scheme, one has to define *where* to refine and *how* to refine. Because of its very high speed, classic h-refinement (Löhner and Baum (1992)) is used to subdivide background grid elements. In this way the possible interactivity on current workstations is maintained.

The selection as to where to refine the background mesh is made with the following assumptions:

(a) points have already been generated;

(b) at each of these points, a value for the characteristic or desired element size δ is given;

(c) for each of these points, the background grid element containing it is known;

(d) a desired increase factor c_i between elements is known.

The refinement selection is then made in two passes (see Figure 3.7).

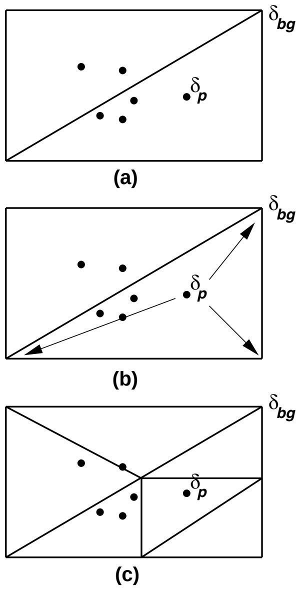


Figure 3.7. Background grid refinement: (a) generated surface points; (b) adjust background grid element size to surface point size; (c) refine background grid as required

Pass 1: Background grid adjustment
Suppose a background grid element has very large element sizes defined at its nodes. If it contains a generated point with characteristic length that is much smaller, an incompatibility is present. The aim of this first pass is to prevent these incompatibilities by comparing lengths. Given a set of two points with coordinates $\mathbf{x}_1$, $\mathbf{x}_2$, as well as an element length parameter δ_1, the maximum allowable element length at the second point may be determined from the geometric progression formula

$$s_n = |\mathbf{x}_2 - \mathbf{x}_1| = \delta_1 \frac{c_i^{n+1} - 1}{c_i - 1}, \tag{3.16}$$

implying

$$\delta_2^* = \delta_1 c_i^n = \frac{s_n(c_i - 1) + \delta_1}{c_i}. \tag{3.17}$$

Given this constraint, one compares the characteristic length of each of the generated points with the element size defined at each of the background grid nodes of the background grid element containing it:

$$\delta_{bg} = \min(\delta_{bg}, \delta_p^*(\mathbf{x}_{bg} - \mathbf{x}_p)). \tag{3.18}$$

Pass 2: Selection of elements to be refined
After the element lengths at the points of the background grid have been made compatible with the lengths of the actual points, the next step is to decide where to refine the background grid. The argument used here is that if there is a significant difference between the element size at generated points and the points of the background grid, the element should be refined. This simple criterion is expressed as

$$\min_{a,b,c,d}(\delta_{bg}) > c_f\delta_p \Rightarrow \text{ refine} \tag{3.19}$$

where, typically, $1.5 \le c_f \le 3$. All elements flagged by this last criterion are subdivided further via classic h-refinement (Löhner and Baum (1992)), and the background grid variables are interpolated linearly for the newly introduced points.

3.2.8. SURFACE GRIDDING WITH ADAPTIVE BACKGROUND GRIDS

Background grid adaptation may be used to automatically generate grids that represent the surface within a required or prescribed accuracy. Consider, as a measure for surface accuracy, the angle variation between two adjacent faces.

With the notation defined in Figure 3.8, the angle between two faces is given by

$$\frac{h}{2r} = \tan\left(\frac{\alpha}{2}\right). \tag{3.20}$$

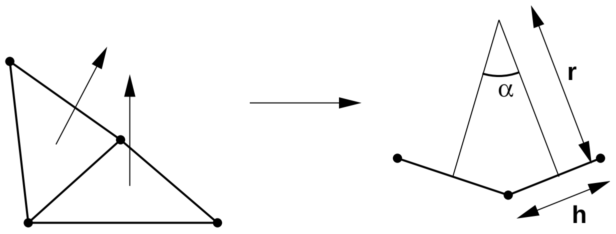


Figure 3.8. Measuring surface curvature

This implies that, for a given element size h_g and angle α_g, the element size for a prescribed angle α_p should be

$$h_p = h_g\frac{\tan(\alpha_p/2)}{\tan(\alpha_g/2)}. \tag{3.21}$$

For other measures of surface accuracy, similar formulae will be encountered. Given a prescribed angle α_p, the point distances for the given surface triangulation are compared to those obtained from (3.21) and reduced appropriately:

$$\delta_i = \min(\delta_i, h_p). \tag{3.22}$$

Surface curvature by itself cannot detect other cases where locally a fine mesh is required. A typical case is shown in Figure 3.9, where a surface patch has some very small side segments, as well as lines with a very small distance between them. This surface patch may belong to a plane, implying that the criteria based on surface curvature will not enforce small elements.

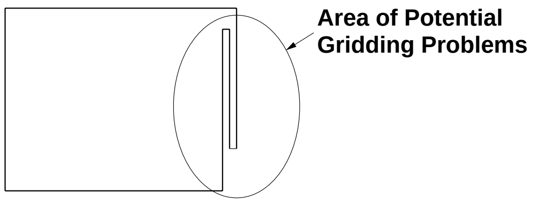


Figure 3.9. Planar surface patch with close lines

Left untreated, even if the surface grid generator were able to generate a mesh, the quality of these surface elements would be such that the resulting tetrahedra would be of poor quality. Therefore, in order to avoid elements of bad quality, the recourse taken is to analyse the proximity of the line in the surface patch, and then reduce the distance parameter for the points δ_i accordingly. These new point distances are then used to further adjust and/or refine the background grid, and a new surface triangulation is generated. This process is repeated until the surface representation is accurate enough.

Adaptive background grids combine a number of advantages:

- possible use in combination with CAD-based element size specification and/or background sources;
- automatic gridding to specified surface deviation tolerance;
- automatic gridding to specified number of elements in gaps;
- smooth transition between surface faces of different size;
- smooth transition from surface faces to volume-elements.

Thus, they may be used to arrive at automatic, minimal-input grid generators.

3.3. Element type

Almost all current unstructured grid generators can only generate triangular or tetrahedral elements. If quad-elements in two dimensions are required, they can either be generated using an advancing front (Zhu *et al.* (1991)) or paving (Blacker and Stephenson (1992)) technique, or by first generating a grid of triangles that is modified further (Rank *et al.* (1993)). This last option can be summarized in the following five algorithmic steps.

Q1. Generate a triangular mesh with elements whose sides are twice as long as the ones of the quad-elements required.

Q2. Fuse as many pairs of triangles into quads as possible without generating quads that are too distorted. This process will leave some triangles in the domain.

Q3. Improve this mixed mesh of quads and triangles by adding/removing points, switching diagonals and smoothing the mesh.

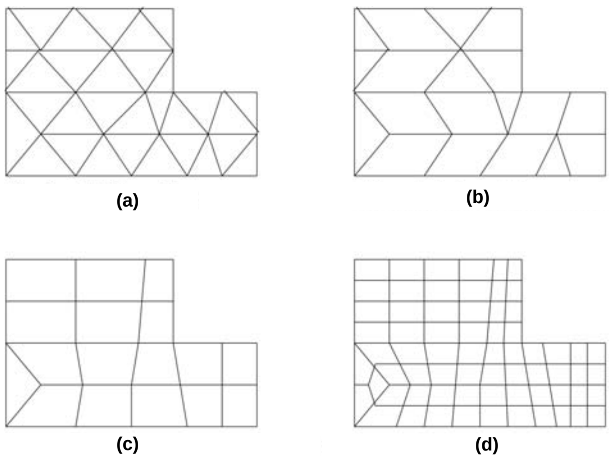


Figure 3.10. Generation of quad grids from triangles: (a) mesh of triangles; (b) after joining triangles into quads; (c) after switching/improvement; (d) after global h-refinement

Q4. H-refine globally the mesh of triangles and quads (see Figure 3.10). In this way, the resulting mesh will only contain quads. Moreover, the quads will now be of the desired size.

Q5. Smooth the final mesh of quads.

Some of the possible operations required to improve a mixed mesh have been summarized in Figures 3.11(a)–(c).

The procedure outlined earlier will not work in three dimensions, the problem being the face-diagonals that appear when tetrahedra are matched with brick elements (see Figure 3.12). The face disappears in two dimensions (only edges are present), making it possible to generate quad-elements from triangles.

3.4. Automatic grid generation methods

There appear to be only the two following ways to fill space with an unstructured mesh.

M1. *Fill empty, i.e. not yet gridded space*. The idea here is to proceed into as yet ungridded space until the complete computational domain is filled with elements. This has been shown diagrammatically in Figure 3.13.

The 'front' denotes the boundary between the region in space that has been filled with elements and that which is still empty. The key step is the addition of a new volume or *element* to the ungridded space. Methods falling under this category are called *advancing front techniques (AFTs)*.

M2. *Modify and improve an existing grid*. In this case, an existing grid is modified by the introduction of new points. After the introduction of each point, the grid is reconnected or reconstructed locally in order to improve the mesh quality. This procedure has been sketched in Figure 3.14.

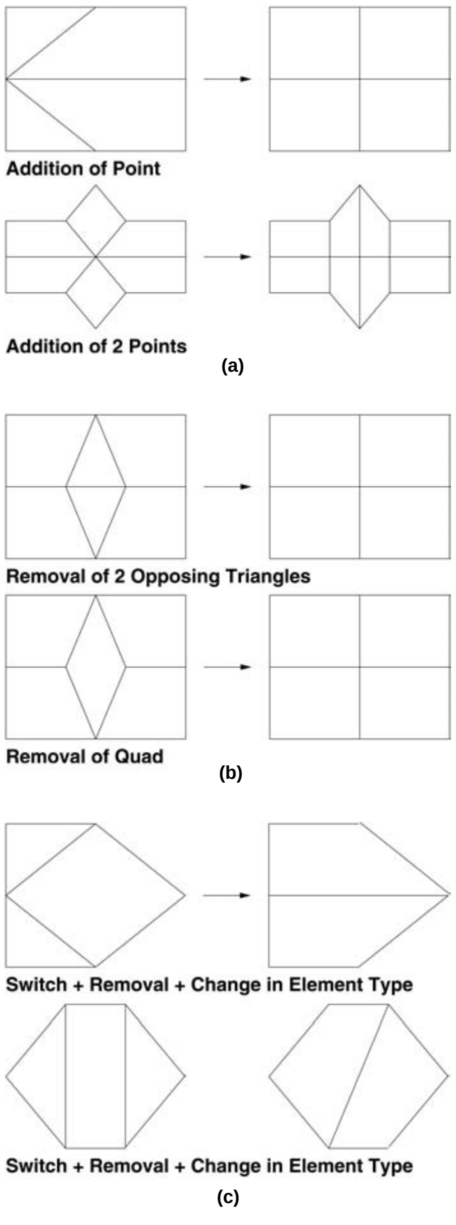

Addition of Point
Addition of 2 Points
(a)
Removal of 2 Opposing Triangles
Removal of Quad
(b)
Switch + Removal + Change in Element Type
Switch + Removal + Change in Element Type
(c)

Figure 3.11. (a) Improvement by addition of points + edge switch; (b) improvement by removal of points + edge switch; and (c) improvement by edge switches

The key step is the addition of a new *point* to an existing grid. The elements whose circumcircle or circumsphere contain the point are removed, and the resulting void faces reconnected to the point, thus forming new elements. The methods falling under this category are called *Delaunay triangulation techniques (DTTs)*.

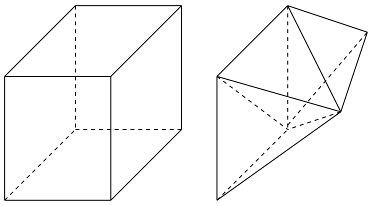


Figure 3.12. Compatibility problems for bricks

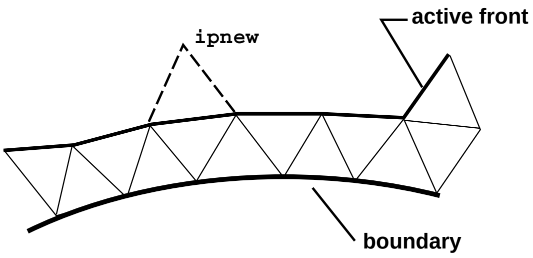


Figure 3.13. The advancing front method

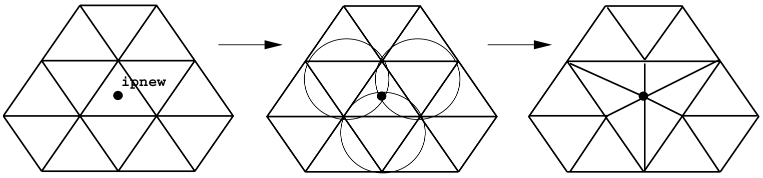


Figure 3.14. The Delaunay triangulation technique

Given the number of ways for specifying element size and shape in space, as well as the method employed to generate the mesh, a number of combinations can be envisioned: AFT and internal measures of grid/front quality (Huet (1990)), DTT and internal measures of grid quality (Holmes and Snyder (1988)), AFT and boxes (van Phai (1982), Lo (1985)), DTT and boxes (Cavendish (1974), Jameson *et al.* (1986), Baker (1989), Yerry and Shepard (1984), Shepard and Georges (1991)), AFT and background grids (Peraire *et al.* (1987), Löhner (1988a,b), Löhner and Parikh (1988), Jin and Tanner (1993), Frykestig (1994)), DTT and background grids (Weatherill (1992)), AFT and sources (Löhner (1993)), DTT and sources (Weatherill (1992)), etc.

Before describing the two most commonly used unstructured grid generation procedures in CFD, the AFT and the DTT, a brief summary of other possible methods is given.

3.5. Other grid generation methods

A number of other methods of generating meshes that are specially suited to a particular application have been developed. If the approximate answer to the problem being simulated is known (say we want to solve the same wing time and time again), the specialized development of an optimal mesh makes good sense. In many of these cases (e.g. an O-mesh for a subsonic/transonic inviscid steady-state airfoil calculation), grids generated by the more general methods listed above will tend to be larger than the specialized ones for the same final accuracy. The main methods falling under this specialized category are as follows.

(a) *Simple mappings*. In this case, it is assumed that the complete computational domain can be mapped into the unit square or cube. The distribution of points and elements in space is controlled either by an algebraic function, or by the solution of a partial differential equation in the transformed space (Thompson *et al.* (1985)). Needless to say, the number of points on opposing faces of the mapped quad or cube have to match line by line.

(b) *Macro-element approach*. Here, the previous approach is applied on a local level by first manually or semi-manually discretizing the domain with large elements. These large elements are subsequently divided up into smaller elements using simple mappings (Zienkiewicz and Phillips (1971)) (see Figure 3.15).

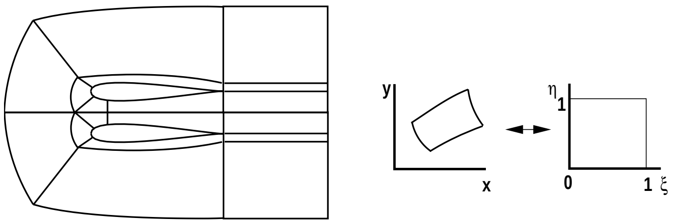


Figure 3.15. The macro-element gridding technique

In the aerospace community, this approach has been termed ‘multi-block’, and a whole service industry dedicated to the proper construction of grids has evolved (Thompson (1988), Steinbrenner *et al.* (1990), Allwright (1990), Marchant and Weatherill (1993), Eiseman (1996)). The macro-element approach is very general, and has the advantage of being well suited for the generation of brick elements, as well as stretched elements to resolve boundary layers. However, it is extremely labour intensive, and efforts to automate the subdivision of space into macro-elements that precedes the actual grid generation have only shown limited success to date (Allwright (1990)). Generating these macro-blocks for complex geometries demands many man-hours. An often cited ‘war story’ has it that it took a team of six engineers half a year in the mid-1990s to produce an acceptable grid for the flowfield around an F-16 fighter with two missiles. Even the emergence of user-friendly software and powerful graphics cards over the last decade has not alleviated the basic shortcoming of the technique, which is hampered by the fact that the automatic generation of high-quality, coarse hex-grids for arbitrary geometries remains an unsolved problem.

(c) *Uniform background grid*. For problems that require uniform grids (e.g. radar cross-section calculations, acoustics, homogeneous turbulence), most of the mesh covering the computational domain can readily be obtained from a uniform grid. This grid is also called a background grid, because it is laid over the computational domain. At the boundaries, the regular mesh or the flow code have to be modified in order to take into account the boundaries. Options here are as follows.

- No modifications. This is quite often done when the grids employed are very fine. The resulting grid exhibits staircasing at the boundaries (see Figure 3.16(a)), and for this reason codes that operate in such a manner are often denoted as 'Legoland' codes.
- Modifying the flow solver. In this case, the change in volumes for the elements cut by the boundaries is taken into account. Other modifications are required to avoid the timestep limitations imposed by very small cells (Pember *et al.* (1995), Landsberg and Boris (1997)).
- Modifying the mesh. Here, the points close to the surfaces are placed on the correct boundary. Some non-trivial logic is required to avoid problems at sharp corners, or when the surface curvature is such that more than one face of any given element is cut by the boundary. After these surface elements have been modified, the mesh is smoothed in order to obtain a more uniform discretization close to the boundaries (Thacker *et al.* (1980), Holmes and Lamson (1986), Schneider (1996, 1997), Taghavi (1996)). The procedure has been sketched in Figure 3.16(b).

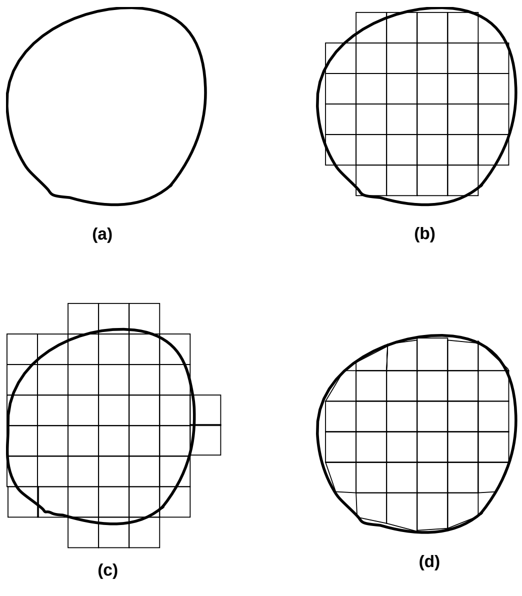


Figure 3.16. Uniform background grids: (a) domain to be gridded; (b) Legoland representation; (c) improved Legoland; (d) modified regular grid

3.6. The advancing front technique

Now that the general strategies currently available to generate unstructured grids have been described, the AFT will be explained in more detail. The aim is to show what it takes to make any of the methods outlined in the previous sections work. Many of the data structures, search algorithms and general coding issues carry over to the other methods. The AFT consists algorithmically of the following steps.

F1. Define the boundaries of the domain to be gridded.

F2. Define the spatial variation of element size, stretchings and stretching directions for the elements to be created. In most cases, this is accomplished with a combination of background grids and sources as outlined above.

F3. Using the information given for the distribution of element size and shape in space and the line definitions, generate sides along the lines that connect surface patches. These sides form an initial front for the triangulation of the surface patches.

F4. Using the information given for the distribution of element size and shape in space, the sides already generated and the surface definition, triangulate the surfaces. This yields the initial front of faces.

F5. Find the generation parameters (element size, element stretchings and stretching directions) for these faces.

F6. Select the next face to be deleted from the front; in order to avoid large elements crossing over regions of small elements, the face forming the smallest new element is selected as the next face to be deleted from the list of faces.

F7. For the face to be deleted:

F7.1 select a 'best point' position for the introduction of a new point `ipnew`;

F7.2 determine whether a point exists in the already generated grid that should be used in lieu of the new point; if there is such a point, set this point to `ipnew` and continue searching (go to F7.2);

F7.3 determine whether the element formed with the selected point `ipnew` crosses any given faces; if it does, select a new point as `ipnew` and try again (go to F7.3).

F8. Add the new element, point and faces to their respective lists.

F9. Find the generation parameters for the new faces from the background grid and the sources.

F10. Delete the known faces from the list of faces.

F11. If there are any faces left in the front, go to F6.

The complete grid generation of a simple 2-D domain using the AFT is shown in Figure 3.17.

In the following, individual aspects of the general algorithm outlined above are treated in more detail.

3.6.1. CHECKING THE INTERSECTION OF FACES

The most important ingredient of the advancing front generator is a reliable and fast algorithm for checking whether two faces intersect. Experience indicates that even slight changes in this portion of the generator greatly influence the final mesh. As with so many other problems in computational geometry, checking whether two faces intersect each other seems trivial for the eye, but is complicated to code. The situation is shown in Figure 3.18.

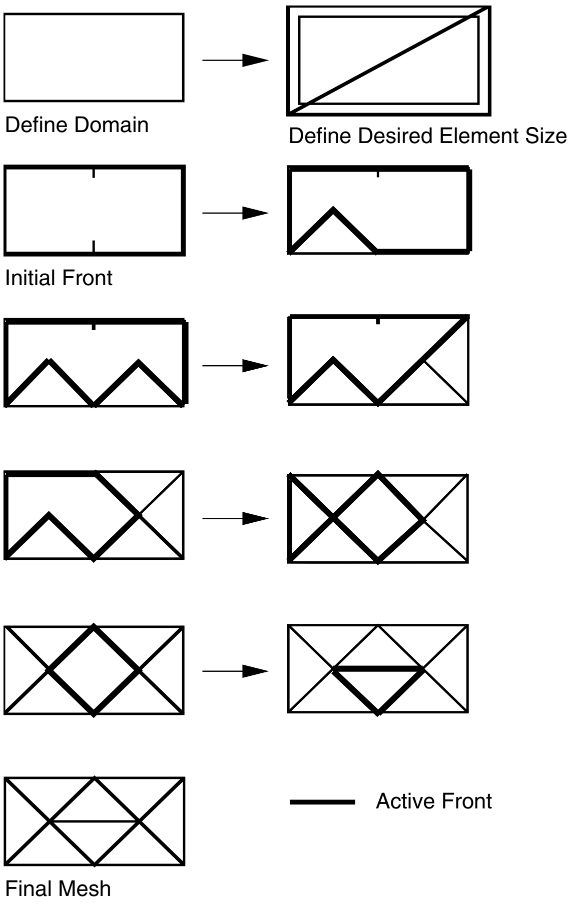


Figure 3.17. The AFT

The checking algorithm is based on the following observation: two triangular faces do not intersect if no side of either face intersects the other face. The idea then is to build all possible side–face combinations between any two faces and check them in turn. If an intersection is found, then the faces cross.

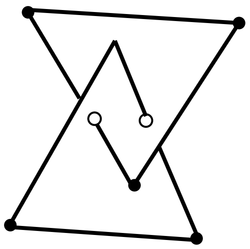


Figure 3.18. Intersection of faces

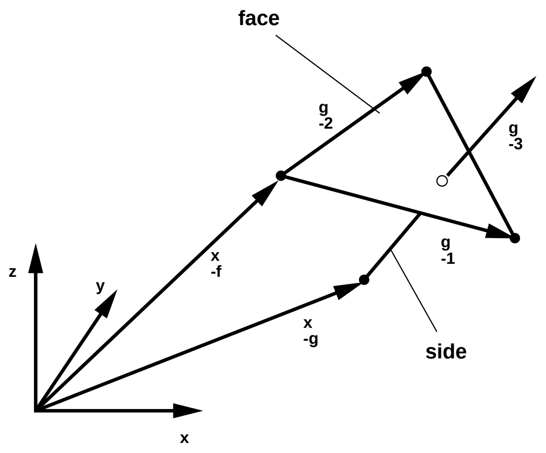


Figure 3.19. Face–side intersection

With the notation defined in Figure 3.19, the intersection point between the plane defined by $\mathbf{x}_f$, $\mathbf{g}_1$, $\mathbf{g}_2$ and the line given by $\mathbf{x}_s$, $\mathbf{g}_3$ is found to be

$$\mathbf{x}_f + \alpha^1\mathbf{g}_1 + \alpha^2\mathbf{g}_2 = \mathbf{x}_s + \alpha^3\mathbf{g}_3, \tag{3.23}$$

where the $\mathbf{g}_i$ vectors form a covariant basis. Using the contravariant basis $\mathbf{g}^i$ defined by

$$\mathbf{g}^i \cdot \mathbf{g}_j = \delta^i_j, \tag{3.24}$$

where δ^i_j denotes the Kronecker delta, the α^i are given by

$$\begin{aligned}\alpha^1 &= (\mathbf{x}_s - \mathbf{x}_f) \cdot \mathbf{g}^1,\\ \alpha^2 &= (\mathbf{x}_s - \mathbf{x}_f) \cdot \mathbf{g}^2,\\ \alpha^3 &= (\mathbf{x}_f - \mathbf{x}_s) \cdot \mathbf{g}^3.\end{aligned} \tag{3.25}$$

The α^1, α^2 correspond to the shape-function values ξ, η of the triangular face (for a more complete description of shape functions, see Chapter 4). The face–side intersection test requires one further quantity, α^4, that is equivalent to the shape function $\zeta = 1 - \xi - \eta$:

$$\alpha^4 = 1 - \alpha^1 - \alpha^2. \tag{3.26}$$

If any of the α^i lie in the interval [0, 1], the side crosses the face. One can extend the region of tolerance, and consider two faces as ‘crossed’ if they only come close together. Then, in order for the side not to cross the face, at least one of the α^i has to satisfy

$$t > \max(-\alpha^i,\ \alpha^i - 1), \quad i = 1, 4, \tag{3.27}$$

where t is a predefined tolerance. By projecting the $\mathbf{g}_i$ onto their respective unit contravariant vectors, one can obtain the actual distance between a face and a side.

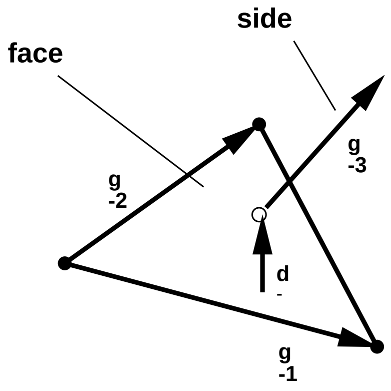


Figure 3.20. Distance of the side from the face

The criterion given by (3.27) would then be replaced by (see Figure 3.20)

$$d > \frac{1}{|\mathbf{g}^i|} \max(-\alpha^i, \alpha^i - 1), \quad i = 1, 4. \tag{3.28}$$

The first form (equation (3.27)) produces acceptable grids. If the face and the side have points in common, then the α^i will all be either 1 or 0. As neither (3.27) nor (3.28) will be satisfied, special provision has to be made for these cases. For each two faces, six side–face combinations are possible. Considering that on average about 40 close faces need to be checked, this method of checking the crossing of faces is very CPU intensive. If coded carelessly, it can easily consume more than 80% of the total CPU required for grid generation. In order to reduce the work load, a three-layered approach can be adopted.

(a) *Min/max search*. The idea here is to disregard all face–face combinations where the distance between faces exceeds some prescribed minimum distance. This can be accomplished by checking the maximum and minimum value for the coordinates of each face. Faces cannot possibly cross each other if, at least for one of the dimensions $i = 1, 2, 3$, they satisfy one of the following inequalities:

$$\max_{\text{face1}}(x^i_A, x^i_B, x^i_C) < \min_{\text{face2}}(x^i_A, x^i_B, x^i_C) - d, \tag{3.29a}$$

$$\min_{\text{face1}}(x^i_A, x^i_B, x^i_C) > \max_{\text{face2}}(x^i_A, x^i_B, x^i_C) + d, \tag{3.29b}$$

where A, B, C denote the corner points of each face.

(b) *Local element coordinates*. The purpose of checking for face crossings is to determine whether the newly formed tetrahedron intersects already given faces. The idea is to extend the previous min/max criterion with the shape functions of the new tetrahedron. If all the points of a given face have shape-function values α^i that have the same sign and lie outside the $[-t,\ 1+t]$ interval, then the tetrahedron cannot possibly cross the face. Such a face is therefore disregarded.

(c) *In-depth analysis of side–face combinations*. All the faces remaining after the filtering process of steps (a) and (b) are analysed using side–face combinations as explained above.

Each of these three filters requires about an order of magnitude more CPU time than the preceding one. When implemented in this way, the face-crossing check requires only 25% of the total grid generation time. On vector machines loops are performed over all the possible combinations, building the $\mathbf{g}_i$, $\mathbf{g}^i$, α^i, etc., in vector mode. Although the vector lengths are rather short, the chaining that results from the lengthy mathematical operations involved results in acceptable speedups on vector machines, as well as low cache misses and high computation-to-memory-fetch ratios on ordinary microchips.

3.6.2. DATA STRUCTURES TO MINIMIZE SEARCH OVERHEADS

The operations that could potentially reduce the efficiency of the algorithm to $O(N^{1.5})$ or even $O(N^2)$ are:

(a) finding the next face to be deleted (step F6);

(b) finding the closest given points to a new point (step F7.2);

(c) finding the faces adjacent to a given point (step F7.3);

(d) finding for any given location the values of generation parameters from the background grid and the sources (steps F5 and F9). This is an interpolation problem on unstructured grids.

The verb 'find' appears in all of these operations. The main task is to design the best data structures for performing the search operations (a)–(d) as efficiently as possible. These data structures are typically binary trees or more complex trees. They were developed in the 1960s for computer science applications. Many variations are possible (see Knuth (1973), Sedgewick (1983)). As with flow solvers, there does not seem to be a clearly defined optimal data structure that all current grid generators use. For each of the data structures currently employed, one can find pathological cases where the performance of the tree search degrades considerably. Data structures that have been used include:

- heap lists to find the next face to be deleted from the front (Williams (1964), Floyd (1964), Knuth (1973), Sedgewick (1983));
- quadtrees (2-D) and octrees (3-D) to locate points that are close to any given location (Knuth (1973), Sedgewick (1983), Yerry and Shepard (1984));
- N-trees, to determine which faces are adjacent to a point.

These data structures have been described in Chapter 2.

3.6.3. ADDITIONAL TECHNIQUES TO INCREASE SPEED

There are some additional techniques that can be used to improve the performance of the advancing front grid generator. The most important of these are listed in the sequel.

(a) *Filtering*. Typically, the number of close points and faces found for checking purposes is far too conservative, i.e. large. As an example, consider the search for close points: there may be up to eight points inside an octant, but of these only one may be close to the face to be taken out. The idea is to filter out these ‘distant’ faces and points in order to avoid extra work afterwards. While the search operations are difficult to vectorize, these filtering operations lend themselves to vectorization in a straightforward way, leading to a considerable overall reduction in CPU requirements. Moreover, filtering requires a modest amount of operations as compared to the work required in subsequent operations. The most important filtering operations are:

- removal of points that are too far away;
- removal of points that are not visible from the face to be removed from the front (these would form elements with negative Jacobians, see Figure 3.21);

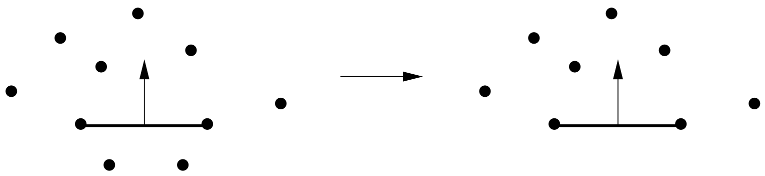


Figure 3.21. Horizon of visibility for face `ifout`

- removal of points that are not visible from the visible adjacent faces to the face to be removed from the front (these would form elements that cross the front, see Figure 3.22);

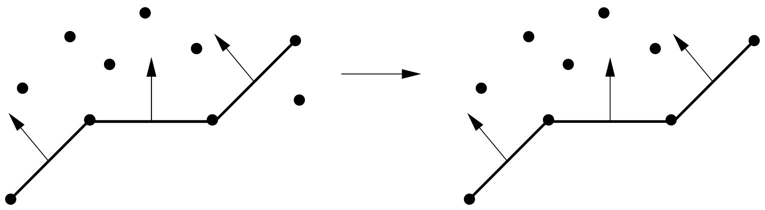


Figure 3.22. Visible adjacent faces to face `ifout`

- removal of faces that cannot see the face to be removed (there is no need to check for these, see Figure 3.23).

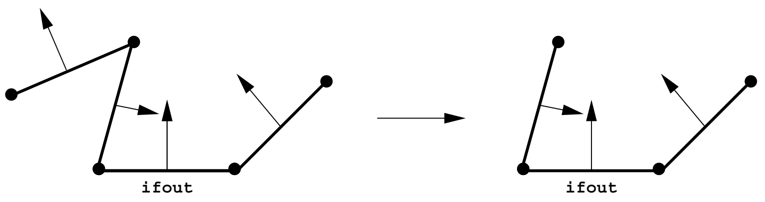


Figure 3.23. Faces that cannot see face `ifout`

(b) *Ordering of close points*. Every point of the list of 'close points' represents a candidate to form a new element. In order not to check in vain points that are unsuitable, the points are ordered according to the quality (and chance of success) of the element they would form. An ordering criterion that has proven very reliable is the ordering according to the angle of the vertex formed with the new points. As can be seen from Figure 3.24, the points with higher angles should be favoured. It is clear that the chances of forming a valid (i.e. non-front-crossing) triangle with the (smaller) angle $\alpha 3$ are much smaller than those of the point with the (larger) angle $\alpha 1$.

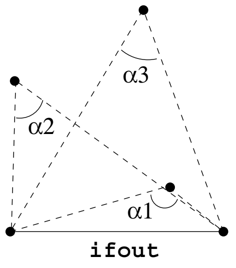


Figure 3.24. Ordering of close points according to angles

(c) *Automatic reduction of unused points*. As the front advances into the domain and more and more tetrahedra are generated, the number of tree levels increases. This automatically implies an increase in CPU time, as more steps are required to reach the lower levels of the trees. In order to reduce this CPU increase as much as possible, all trees are automatically restructured. All points which are completely surrounded by tetrahedra are eliminated from the trees. This procedure has proven to be extremely efficient. It reduces the asymptotic complexity of the grid generator to less than $O(N \log N)$. In fact, in most practical cases one observes a linear $O(N)$ asymptotic complexity, as CPU is traded between subroutine call overheads for small problems and less close faces on average for large problems.

(d) *Global h-refinement*. While the basic advancing front algorithm is a scalar algorithm with a considerable number of operations (search, compare, check), a global refinement of the mesh (so-called h-refinement) requires far fewer operations. Moreover, it can be completely vectorized, and is easily ported to shared-memory parallel machines. Therefore, the grid generation process can be made considerably faster by first generating a coarser mesh that has all the desired variations of element size and shape in space, and then refining globally this first mesh with classic h-refinement (Löhner and Morgan (1987), Löhner (1990b)). Typical speedups achieved by using this approach are 1:6 to 1:7 for each level of global h-refinement.

Typical 3-D advancing front generators construct grids at a rate of 600 000 tetrahedra per minute on a Pentium IV with 3.2 GHz clock speed. With one level of h-refinement, these rates increase by a factor in excess of 1:6. This rate is essentially independent of grid size. However, it may decrease for very small grids.

3.6.4. ADDITIONAL TECHNIQUES TO ENHANCE RELIABILITY

The advancing front algorithm described above may still fail for some pathological cases. The newcomer should be reminded that in three dimensions even the slightest chance of something going awry has to be accounted for. In a mesh of over a million tetrahedra (common even for Euler runs and small by 2008 standards), a one-in-a-million possibility becomes a reality. The following techniques have been found effective in enhancing the reliability of advancing front grid generators to a point where they can be applied on a routine basis in a production environment.

(a) *Avoidance of bad elements*. It is important not to allow any bad elements to be created during the generation process. These bad elements can wreak havoc when trying to introduce further elements at a later stage. Therefore, if a well-shaped tetrahedron cannot be introduced for the current face, the face is skipped.

(b) *Sweep and retry*. If any faces where new elements could not be introduced remain in the field, these regions are enlarged and remeshed again. This 'sweep and retry' technique has proven extremely robust and reliable. It has also made smoothing of meshes possible: if elements with negative or small Jacobians appear during smoothing (as is the case with most spring-analogy smoothers), these elements are removed. The unmeshed regions of space are then regridded. Smoothing improves the mesh quality substantially, leading to better results in field solvers.

3.7. Delaunay triangulation

The DTT has a long history in mathematics, geophysics and engineering (George (1991), George and Borouchaki (1998)). Given a set of points $\mathcal{P} := \mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$, one may define a set of regions or volumes $\mathcal{V} := v_1, v_2, \ldots, v_n$ assigned to each of the points, that satisfy the following property: any location within v_i is closer to $\mathbf{x}_i$ than to any other of the points,

$$v_i := \mathcal{P} : \|\mathbf{x} - \mathbf{x}_i\| < \|\mathbf{x} - \mathbf{x}_j\|, \quad \forall j \neq i. \tag{3.30}$$

This set of volumes $\mathcal{V}$, which covers the domain completely, is known as the *Dirichlet tessellation*. The volumes v_i are convex polyhedra and can assume fascinating shapes. They are referred to as Voronoi regions. Joining all the pairs of points $\mathbf{x}_i, \mathbf{x}_j$ across polyhedral boundaries results in a triangulation of the convex hull of $\mathcal{P}$. It is this triangulation that is commonly known as the *Delaunay triangulation*. The set of triangles (tetrahedra) that form the Delaunay triangulation satisfy the property that no other point is contained within the circumcircle (circumsphere) formed by the nodes of the triangle (tetrahedron). Although this and other properties of Delaunay 3-D triangulations have been studied for some time, practical triangulation procedures have only appeared in the 1990s (Lee and Schachter (1980), Bowyer (1981), Watson (1981), Tanemura *et al.* (1983), Sloan and Houlsby (1984), Cavendish *et al.* (1985), Kirkpatrick (1985), Shenton and Cendes (1985), Baker (1987), Choi *et al.* (1988), Holmes and Snyder (1988), Baker (1989), Mavriplis (1990), George *et al.* (1990), Joe (1991a,b), George (1991), George *et al.* (1991b), George and Hermeline (1992), Weatherill (1992), Weatherill *et al.* (1993a), Müller (1993) Müller *et al.* (1993), Weatherill (1994), Weatherill *et al.* (1994), Boender (1994), Marcum (1995), Barth (1995)). Most of these are based on the Bowyer–Watson algorithm (Bowyer (1981), Watson (1981)), which is summarized here for 3-D applications.

B1. Define the convex hull within which all points will lie. This can either be a single large tetrahedron (four points) or a brick (eight points) subdivided into five tetrahedra.

B2. Introduce a new point $\mathbf{x}_{n+1}$.

B3. Find all tetrahedra `LEDEL(1:NEDEL)` whose circumsphere contains $\mathbf{x}_{n+1}$; these are the tetrahedra that will be deleted. The resulting cavity is called the star-shaped domain.

B4. Find all the points belonging to these tetrahedra.

B5. Find all the external faces `LFEXT(1:NFEXT)` of the void that results due to the deletion of these tetrahedra.

B6. Form new tetrahedra by connecting the `NFEXT` external faces to the new point $\mathbf{x}_{n+1}$.

B7. Add the new elements and the point to their respective lists.

B8. Update all data structures.

B9. If more points are to be introduced, go to B2.

The complete grid generation of a simple 2-D domain using the Delaunay triangulation algorithm is shown in Figure 3.25.

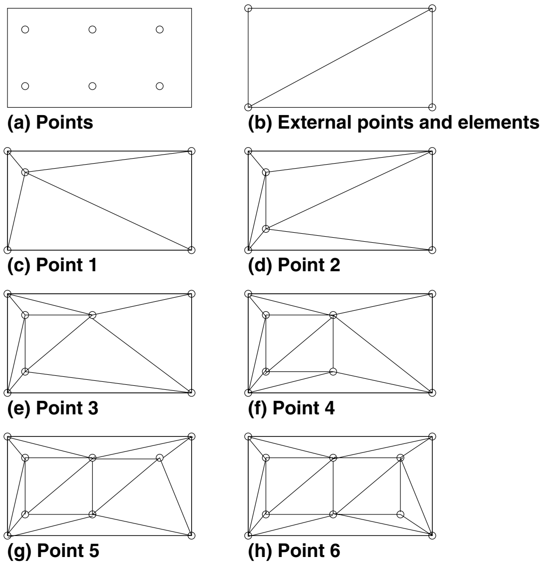


Figure 3.25. The DTT

The algorithm described above assumes a given point distribution. The specification of a point distribution (in itself a tedious task) may be replaced by a general specification of desired element size and shape in space. The key idea, apparently proposed independently by Frey (1987) and Holmes and Snyder (1988), is to check the discrepancy between the desired and the actual element shape and size of the current mesh. Points are then introduced in those regions where the discrepancy exceeds a user-defined tolerance. An automatic Delaunay mesh generator of this kind proceeds as follows.

```
D1. Assume given a boundary point distribution.
D2. Generate a Delaunay triangulation of the boundary points.
D3. Using the information stored on the background grid and the
sources, compute the desired element size and shape for the points
of the current mesh.
D5. nnewp=0                                    ! Initialize new point counter
D6. do ielem=1,nelem                             ! Loop over the elements
        Define a new point inewp at the centroid of ielem;
        Compute the distances dispc(1:4) from inewp to the four nodes
        of ielem;
        Compare dispc(1:4) to the desired element size and shape;
        If any of the dispc(1:4) is smaller than a fraction
        $\alpha$ of the desired element length: skip the element (goto D6);
        Compute the distances dispn(1:nneip) from inewp to the new
        points in the neighbourhood;
        If any of the dispn(1:nneip) is smaller than a fraction
        $\beta$ of the desired element length: skip the element (goto D6);
        nnewp=nnewp+1                              ! Update new point list
        Store the desired element size and shape for the new point;
    enddo
D7. if(nnewp.gt.0) then
        Perform a Delaunay triangulation for the new points;
                                                 goto D.5
    endif
```

The procedure outlined above introduces new points in the elements. One can also introduce them at edges (George and Borouchaki (1998)). In the following, individual aspects of the general algorithm outlined above are treated in more detail.

3.7.1. CIRCUMSPHERE CALCULATIONS

The most important ingredient of the Delaunay generator is a reliable and fast algorithm for checking whether the circumsphere (-circle) of a tetrahedron (triangle) contains the points to be inserted. A point $\mathbf{x}_p$ lies within the radius R_i of the sphere centred at $\mathbf{x}_c$ if

$$d_p^2 = (\mathbf{x}_p - \mathbf{x}_c) \cdot (\mathbf{x}_p - \mathbf{x}_c) < R_i^2. \tag{3.31}$$

This check can be performed without any problems unless $|d_p - R_i|^2$ is of the order of the round-off of the computer. In such a case, an error may occur, leading to an incorrect rejection or acceptance of a point. Once an error of this kind has occurred, it is very difficult to correct, and the triangulation process breaks down. Baker (1987) has determined the following condition:

Given the set of points $\mathcal{P} := \mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$ with characteristic lengths $d_{\max} = \max|\mathbf{x}_i - \mathbf{x}_j| \ \forall i \neq j$ and $d_{\min} = \min|\mathbf{x}_i - \mathbf{x}_j| \ \forall i \neq j$, the floating point arithmetic precision required for the Delaunay test should be better than

$$\epsilon = \left(\frac{d_{\min}}{d_{\max}}\right)^2. \tag{3.32}$$

Consider the generation of a mesh suitable for inviscid flow simulations for a typical transonic airliner (e.g. Boeing-747). Taking the wing chord length as a reference length, the smallest elements will have a side length of the order of $10^{-3}L$, while far-field elements may be located as far as $10^{2}L$ from each other. This implies that $\epsilon = 10^{-10}$, which is beyond the 10^{-8} accuracy of 32-bit arithmetic. For these reasons, unstructured grid generators generally operate with 64-bit arithmetic precision. When introducing points, a check is conducted for the condition

$$|d_p - R_i|^2 < \epsilon, \tag{3.33}$$

where ϵ is a preset tolerance that depends on the floating point accuracy of the machine. If condition (3.33) is met, the point is rejected and stored for later use. This ‘skip and retry’ technique is similar to the ‘sweep and retry’ procedure already described for the AFT. In practice, most grid generators work with double precision and the condition (3.33) is seldom met.

A related problem of degeneracy that may arise is linked to the creation of very flat elements or ‘slivers’ (Cavendish *et al.* (1985)). The calculation of the circumsphere for a tetrahedron is given by the conditions

$$(\mathbf{x}_i - \mathbf{x}_c) \cdot (\mathbf{x}_i - \mathbf{x}_c) = R^2, \quad i = 1, 4, \tag{3.34}$$

yielding four equations for the four unknowns $\mathbf{x}_c$, R. If the four points of the tetrahedron lie on a plane, the solution is impossible ($R \to \infty$). In such a case, the point to be inserted is rejected and stored for later use (skip and retry).

3.7.2. DATA STRUCTURES TO MINIMIZE SEARCH OVERHEADS

The operations that could potentially reduce the efficiency of the algorithm to $O(N^{1.5})$ or even $O(N^2)$ are:

(a) finding all tetrahedra whose circumspheres contain a point (step B3);

(b) finding all the external faces of the void that results due to the deletion of a set of tetrahedra (step B5);

(c) finding the closest new points to a point (step D6);

(d) finding for any given location the values of generation parameters from the background grid and the sources (step D3).

The verb ‘find’ appears in all of these operations. The main task is to design the best data structures for performing the search operations (a)–(d) as efficiently as possible. As before, many variations are possible here, and some of these data structures have already been discussed for the AFT. The principal data structure required to minimize search overheads is the ‘element adjacent to element’ or ‘element surrounding element’ structure `esuel(1:nfael,1:nelem)` that stores the neighbour elements of each element. This structure, which was already discussed in Chapter 2, is used to march quickly through the grid when trying to find the tetrahedra whose circumspheres contain a point. Once a set of elements has been marked for removal, the outer faces of this void can be obtained by interrogating `esuel`. As the new points to be introduced are linked to the elements of the current mesh (step D6), `esuel` can also be used to find the closest new points to a point. Furthermore, the equivalent `esuel` structure for the background grid can be used for fast interpolation of the desired element size and shape.

3.7.3. BOUNDARY RECOVERY

A major assumption that is used time and again to make the Delaunay triangulation process both unique and fast is the Delaunay property itself: namely, that no other point should reside in the circumsphere of any tetrahedron. This implies that in general during the grid generation process some of the tetrahedra will break through the surface. The result is a mesh that satisfies the Delaunay property, but is not surface conforming (see Figure 3.26).

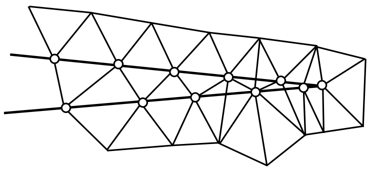


Figure 3.26. Non-body conforming Delaunay triangulation

In order to recover a surface conforming mesh, a number of techniques have been employed.

(a) *Extra point insertion*. By inserting points close to the surface, one can force the triangulation to be surface conforming. This technique has been used extensively by Baker (1987, 1989) for complete aircraft configurations. It can break down for complex geometries, but is commonly used as a 'first cut' approach within more elaborate approaches.

(b) *Algebraic surface recovery*. In this case, the faces belonging to the original surface point distribution are tested one by one. If any tetrahedron crosses these faces, local reordering is invoked. These local operations change the connectivity of the points in the vicinity of the face in order to arrive at a surface-conforming triangulation. The complete set of possible transformations can be quite extensive. For this reason, surface recovery can take more than half of the total time required for grid generation using the Delaunay technique (Weatherill (1992), Weatherill *et al.* (1993a)).

3.7.4. ADDITIONAL TECHNIQUES TO INCREASE SPEED

There are some additional techniques that can be used to improve the performance of the Delaunay grid generator. The most important of these are the following.

(a) *Point ordering*. The efficiency of the DTT is largely dependent on the amount of time taken to find the tetrahedra to be deleted as a new point is introduced. For the main point introduction loop over the current set of tetrahedra, these elements can be found quickly from each current element and the neighbour list `esuel`. It is advisable to order the first set of boundary points so that contiguous points in the list are neighbours in space. Once such an ordering has been achieved, only a local set of tetrahedra needs to be inspected. After one of the tetrahedra to be eliminated has been found, the rest are again determined via `esuel`.

(b) *Vectorization of background grid/source information*. During each pass over the elements introducing new points, the distance from the element centroid to the four nodes is compared with the element size required from the background grid and the sources. These distances may all be computed at the same time, enabling vectorization and/or parallelization on shared-memory machines. After each pass, the distance required for the new points is again computed in vector/parallel mode.

(c) *Global h-refinement*. While the basic Delaunay algorithm is a scalar algorithm with a considerable number of operations (search, compare, check), a global refinement of the mesh (so-called h-refinement) requires far fewer operations. Moreover, it can be completely vectorized, and is easily ported to shared-memory parallel machines. Therefore, the grid generation process can be made considerably faster by first generating a coarser mesh that has all the desired variations of element size and shape in space, and then refining globally this first mesh with classic h-refinement. Typical speedups achieved by using this approach are 1:6 to 1:7 for each level of global h-refinement.

(d) *Multiscale point introduction*. The use of successive passes of point generation as outlined above automatically results in a 'multiscale' point introduction. For an isotropic mesh, each successive pass will result in five to seven times the number of points of the previous pass. In some applications, *all* point locations are known before the creation of elements begins. Examples are remote sensing (e.g. drilling data) and Lagrangian particle or particle–finite element method solvers (Idelsohn *et al.* (2003)). In this case, a spatially ordered list of points for the fine mesh will lead to a large number of faces in the 'star-shaped domain' when elements are deleted and re-formed. The case shown in Figure 3.27 may be exaggerated by the external shape of the domain (for some shapes and introduction patterns, *nearly all* elements can be in the star-shaped domain). The main reason for the large number of elements treated, and hence the inefficiency, is the large discrepancy in size 'before' and 'after' the last point introduced. In order to obtain similar element sizes throughout the mesh, and thus near-optimal efficiency, the points are placed in a bin. Points are then introduced by considering, in several passes over the mesh, every eighth, fourth, second, etc., bin in each spatial dimension until the list of points is exhausted.

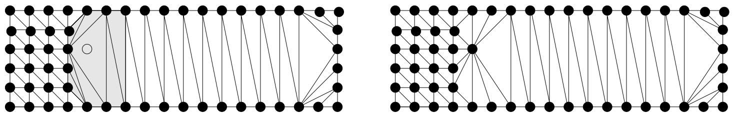


Figure 3.27. Large number of elements in a star-shaped domain

Sustained speeds in excess of 250 000 tetrahedra per minute have been achieved on the Cray-YMP (Weatherill (1992), Weatherill and Hassan (1994), Marcum (1995)), and the procedure has been ported to parallel machines (Weatherill (1994)). In some cases, the Delaunay circumsphere criterion is replaced by or combined with a min(max) solid angle criterion (Joe (1991a,b), Barth (1995), Marcum and Weatherill (1995b)), which has been shown to improve the quality of the elements generated. For these techniques, a 3-D edge-swapping technique is used to speed up the generation process.

3.7.5. ADDITIONAL TECHNIQUES TO ENHANCE RELIABILITY AND QUALITY

The Delaunay algorithm described above may still fail for some pathological cases. The following techniques have been found effective in enhancing the reliability of Delaunay grid generators to a point where they can be applied on a routine basis in a production environment.

(a) *Avoidance of bad elements*. It is important not to allow any bad elements to be created during the generation process. These bad elements can wreak havoc when trying to introduce further points at a later stage. Therefore, if the introduction of a point creates bad elements, the point is skipped. The quality of an element can be assessed while computing the circumsphere.

(b) *Consistency in degeneracies*. The Delaunay criterion can break down for some ‘degenerate’ point distributions. One of the most common degeneracies arises when the points are distributed in a regular manner. If five or more points lie on a sphere (or four or more points lie on a circle in two dimensions), the triangulation is not unique, since the ‘inner’ connections between these points can be taken in a variety of ways. This and similar degeneracies do not present a problem as long as the decision as to whether a point is inside or outside the sphere is consistent for all the tetrahedra involved.

(c) *Front-based point introduction*. When comparing 2-D grids generated by the AFT or the DTT, the most striking difference lies in the appearance of the grids. The Delaunay grids always look more ‘ragged’ than the advancing front grids. This is because the grid connectivity obtained from Delaunay triangulations is completely free, and the introduction of points in elements does not allow a precise control. In order to improve this situation, several authors (Merriam (1991), Mavriplis (1993), Müller *et al*. (1993), Rebay (1993), Marcum (1995)) have tried to combine the two methods. These methods are called advancing front Delaunay, and can produce extremely good grids that satisfy the Delaunay or min(max) criterion.

(d) *Beyond Delaunay*. The pure Delaunay circumsphere criterion can lead to a high percentage of degenerate elements called ‘slivers’. In two dimensions the probability of bad elements is much lower than in three dimensions, and for this reason this shortcoming was ignored for a while. However, as 3-D grids became commonplace, the high number of slivers present in typical Delaunay grids had to be addressed. The best way to avoid slivers is by relaxing the Delaunay criterion. The star-shaped domain is modified by adding back elements whose faces would lead to bad elements. This is shown diagrammatically in Figure 3.28. The star-shaped domain, which contains element A–C–B, would lead, after reconnection, to the bad (inverted) element A–B–P. Therefore, element A–C–B is removed from the star-shaped domain, and added back to the mesh before the introduction of the new point P. This fundamental departure from the traditional Delaunay criterion, first proposed by George *et al*. (1990) to the chagrin of many mathematicians and computational geometers, has allowed this class of unstructured grid generation algorithms to produce reliably quality grids. It is a simple change, but has made the difference between a theoretical exercise and a practical tool.

3.8. Grid improvement

Practical implementations of either advancing front or Voronoi/Delaunay grid generators indicate that in certain regions of the mesh abrupt variations in element shape or size may be present. These variations appear even when trying to generate perfectly uniform grids. The reason is a simple one: the perfect tetrahedron is not a space-filling polyhedron. In order

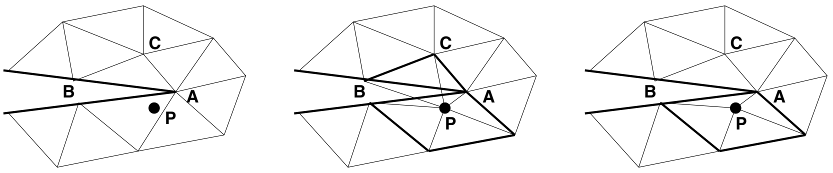


Figure 3.28. The modified Delaunay algorithm

to circumvent any possible problems these irregular grids may trigger for field solvers, the generated mesh is optimized further in order to improve the uniformity of the mesh. The most commonly used ways of mesh optimization are:

(a) removal of bad elements;

(b) Laplacian smoothing;

(c) functional optimization;

(d) selective mesh movement; and

(e) diagonal swapping.

3.8.1. REMOVAL OF BAD ELEMENTS

The most straightforward way to improve a mesh containing bad elements is to get rid of them. For tetrahedral grids this is particularly simple, as the removal of an internal edge does not lead to new element types for the surrounding elements. Once the bad elements have been identified, they are compiled into a list and interrogated in turn. An element is removed by collapsing the points of one of the edges, as shown in Figure 3.29.

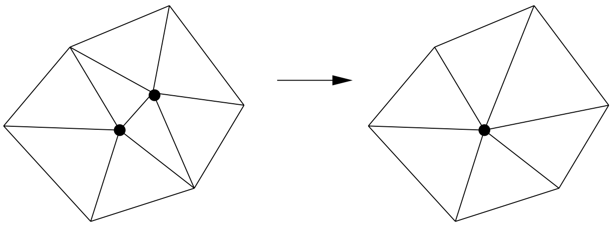


Figure 3.29. Element removal by edge collapse

This operation also removes all the elements that share this edge. It is advisable to make a check of which of the points of the edge should be kept: point 1, point 2 or a point somewhere on the edge (e.g. the mid-point). This implies checking all elements that contain either point 1 or point 2. This procedure of removing bad elements is simple to implement and relatively fast. On the other hand, it can only improve mesh quality to a certain degree. It is therefore used mainly in a pre-smoothing or pre-optimization stage, where its main function is to eradicate from the mesh elements of very bad quality.

3.8.2. LAPLACIAN SMOOTHING

A number of smoothing techniques are lumped under this name. The edges of the triangulation are assumed to represent springs. These springs are relaxed in time using an explicit timestepping scheme, until an equilibrium of spring forces has been established. Because ‘globally’ the variations of element size and shape are smooth, most of the non-equilibrium forces are local in nature. This implies that a significant improvement in mesh quality can be achieved rather quickly. The force exerted by each spring is proportional to its length and is along its direction. Therefore, the sum of the forces exerted by all springs surrounding a point can be written as

$$\mathbf{f}_i = c \sum_{j=1}^{ns_i} (\mathbf{x}_j - \mathbf{x}_i), \tag{3.35}$$

where c denotes the spring constant, $\mathbf{x}_i$ the coordinates of the point and the sum extends over all the points surrounding the point. The time advancement for the coordinates is accomplished as follows:

$$\Delta\mathbf{x}_i = \Delta t \frac{1}{ns_i} \mathbf{f}_i. \tag{3.36}$$

At the surface of the computational domain, no movement of points is allowed, i.e. $\Delta\mathbf{x} = 0$. Usually, the timestep (or relaxation parameter) is chosen as $\Delta t = 0.8$, and five to six timesteps yield an acceptable mesh. The application of the Laplacian smoothing technique can result in inverted or negative elements. The presence of even one element with a negative Jacobian will render most field solvers inoperable. Therefore, these negative elements are eliminated. For the AFT, it has been found advisable to remove not only the negative elements, but also all elements that share points with them. This element removal gives rise to voids or holes in the mesh, which are regridded using the AFT. Another option, which can also be used for the Delaunay technique, is the removal of negative elements using the techniques described before.

3.8.3. GRID OPTIMIZATION

Another way to improve a given mesh is by writing a functional whose magnitude depends on the discrepancy between the desired and actual element size and shape (Cabello *et al.* (1992)). The minimization of this functional, whose value depends on the coordinates of the points, is carried out using conventional minimization techniques. These procedures represent a sophisticated mesh movement strategy.

3.8.4. SELECTIVE MESH MOVEMENT

Selective mesh movement tries to improve the mesh quality by performing a local movement of the points. If the movement results in an improvement of mesh quality, the movement is kept. Otherwise, the old point position is retained. The most natural way to move points is along the directions of the edges touching them.

With the notation of Figure 3.30, point i is moved in the direction $\mathbf{x}^j - \mathbf{x}^i$ by a fraction of the edge length, i.e.

$$\Delta\mathbf{x} = \pm\alpha(\mathbf{x}^j - \mathbf{x}^i). \tag{3.37}$$

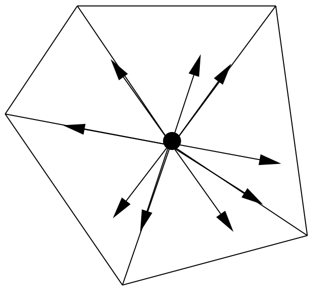


Figure 3.30. Mesh movement directions

After each of these movements, the quality of each element containing point i is checked. Only movements that produce an improvement in element quality are kept. The edge fraction α is diminished for each pass over the elements. Typical values for α are $0.02 \leq \alpha \leq 0.10$, i.e. the movement does not exceed 10% of the edge length. This procedure, while general, is extremely expensive for tetrahedral meshes. This is because, for each pass over the mesh, we have approximately seven edges for each point, i.e. 14 movement directions and approximately 22 elements (4 nodes per element, 5.5 elements per point) surrounding each point to be evaluated for each of the movement directions, i.e. approximately 308*`NPOIN` elements to be tested. To make matters worse, the evaluation of element quality typically involves arc-cosines (for angle evaluations), which consume a large amount of CPU time. The main strength of selective mesh movement algorithms is that they remove efficiently very bad elements. They are therefore used only for points surrounded by bad elements, and as a post-smoothing procedure.

3.8.5. DIAGONAL SWAPPING

Diagonal swapping attempts to improve the quality of the mesh by reconnecting locally the points in a different way (Freitag and Gooch (1997)). Examples of possible 3-D swaps are shown in Figures 3.31 and 3.32.

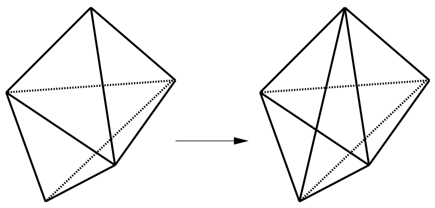


Figure 3.31. Diagonal swap case 2:3

An optimality criterion that has proven reliable is the one proposed by George and Borouchaki (1998),

$$Q = \frac{h_{\max} S}{V}, \tag{3.38}$$

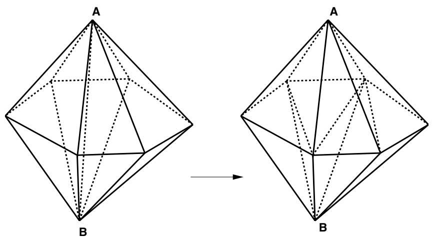


Figure 3.32. Diagonal swap case 6:8

where $h_{\max}$, S and V denote the maximum edge length, total surface area and volume of a tetrahedron. The number of cases to be tested can grow factorially with the number of elements surrounding an edge. Figure 3.33 shows the possibilities to be tested for four, five and six elements surrounding an edge. Note the rapid (factorial) increase of cases with the number of edges.

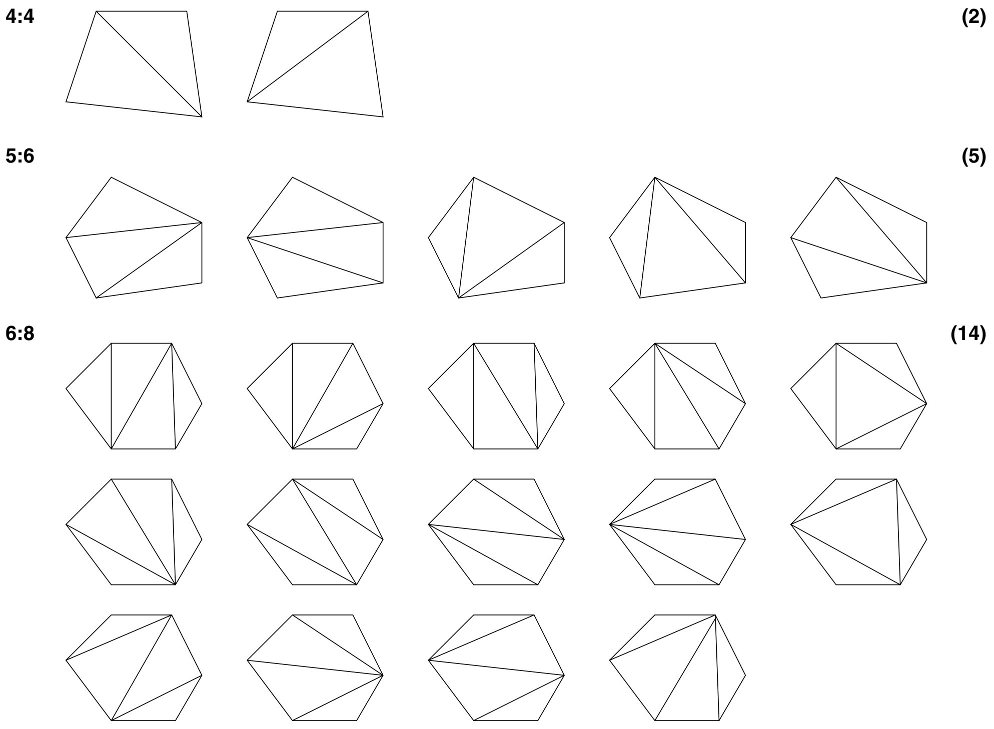


Figure 3.33. Swapping cases

Given that these tests are computationally intensive, considerable care is required when coding a fast diagonal swapper. Techniques that are commonly used include:

- treatment of bad ($Q > Q_{\text{tol}}$), untested elements only;

- processing of elements in an ordered way, starting with the worst (highest chance of reconnection);
- rejection of bad combinations at the earliest possible indication of worsening quality;
- marking of tested and unswapped elements in each pass.

3.9. Optimal space-filling tetrahedra

Unlike the optimal (equilateral) triangle, the optimal (equilateral) tetrahedron shown in Figure 3.34 is not space-filling. All the edge angles have $\alpha = 70.52°$, a fact that does not permit an integer division of the 360° required to surround any given edge. Naturally, the question arises as to which is the optimal space-filling tetrahedron (Fuchs (1998), Naylor (1999), Bridson *et al.* (2005)).

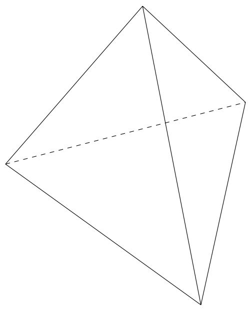


Figure 3.34. An ideal (equilateral) tetrahedron

One way to answer this question is to consider the deformation of a cube split into tetrahedra as shown in Figure 3.35. The tetrahedra are given by: `tet1=1,2,4,5`, `tet2=2,4,5,6`, `tet3=4,8,5,6`, `tet4=2,3,4,6`, `tet5=4,3,8,6`, `tet6=3,7,8,6`. This configuration is subjected to an affine transformation, whereby faces `4,3,7,8` and `5,6,7,8` are moved with an arbitrary translation vector. The selection of the prisms in the original cube retains generality by using arbitrary translation vectors for the face movement. In order to keep face `1,2,4,3` in the x, y plane, no movement is allowed in the z-direction for face `4,3,7,8`. Since the faces remain plane, the configuration is space-filling (and hence so are the tetrahedra).

The problem may be cast as an optimization problem with five unknowns, with the aim of maximizing/minimizing quality criteria for the tetrahedra obtained. Typical quality criteria include:

- equidistance of sides;
- maximization of the minimum angle;
- equalization of all angles;
- George's $h_{\min}$ Area/Volume criterion (equation (3.38)).

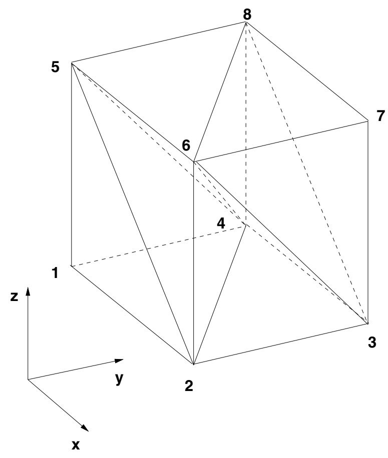


Figure 3.35. A cube subdivided into tetrahedra

An alternative is to invoke the argument that any space-filling tetradron must be self-similar when refined. In this way, the tetrahedron can be regarded as coming from a previously refined tetrahedron, thus filling space. If we consider the refinement configuration shown in Figure 3.36, the displacements in x, y of point 3 and the displacements in x, y, z of point 4 may be regarded as the design variables, and the problem can again be cast as an optimization problem.

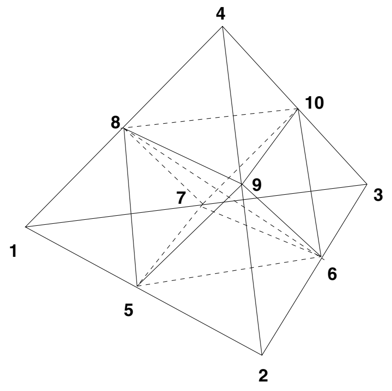


Figure 3.36. H-refinement of a tetrahedron

As expected, both approaches yield the same optimal space-filling tetrahedron, given by:

$$l_{\min} = 1.0, l_{\max} = 1.157,$$

$$\alpha_1 = \alpha_2 = \alpha_3 = \alpha_4 = 60.00°,$$

$$\alpha_5 = \alpha_6 = 90.00°.$$

One can show that this tetrahedron corresponds to the Delaunay triangulation (tetrahedrization) of the points of a body-centred cubic (BCC) lattice given by two Cartesian point distributions that have been displaced by $(\Delta x/2, \Delta y/2, \Delta 2/2)$ (see Figure 3.37). Following Naylor (1999) this tetrahedron will be denoted as an *isotet*.

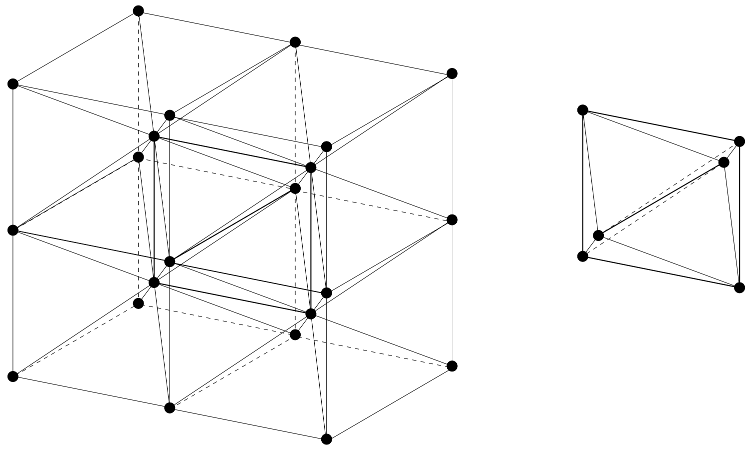


Figure 3.37. BCC lattice

3.10. Grids with uniform cores

The possibility to create near-optimal space-filling tetrahedra allows the generation of grids where the major portion of the volume is composed of such near-perfect elements, and the approximation to a complex geometry is accomplished by a relatively small number of 'truly unstructured' elements. These types of grids are highly suitable for wave propagation problems (acoustics, electromagnetics), where mesh isotropy is required to obtain accurate results. The generation of such grids is shown in Figure 3.38.

In a first step the surface of the computational domain is discretized with triangles of a size as prescribed by the user. As described above, this is typically accomplished through a combination of background grids, sources and element sizes linked to CAD entities. In a second step a mesh of space-filling tetrahedra (or even a Cartesian mesh subdivided into tetrahedra) that has the element size of the largest element desired in the volume is superimposed onto the volume. From this point onwards this 'core mesh' is treated as an unstructured mesh. This mesh is then adaptively refined locally so as to obtain the element size distribution prescribed by the user. Once the adaptive core mesh is obtained, the elements that are outside the domain to be gridded are removed. A number of techniques have been tried to make this step both robust and fast. One such technique uses a fine, uniform voxel (bin, Cartesian) mesh that covers the entire computational domain. All voxels that are crossed by the surface triangulation are marked. A marching cube (advancing layers) technique is then used to mark all the voxels that are inside/outside the computational domain. Any element of the adaptive Cartesian mesh that covers a voxel marked as either crossed by the surface triangulation or outside the computational domain is removed. This yields an additional list

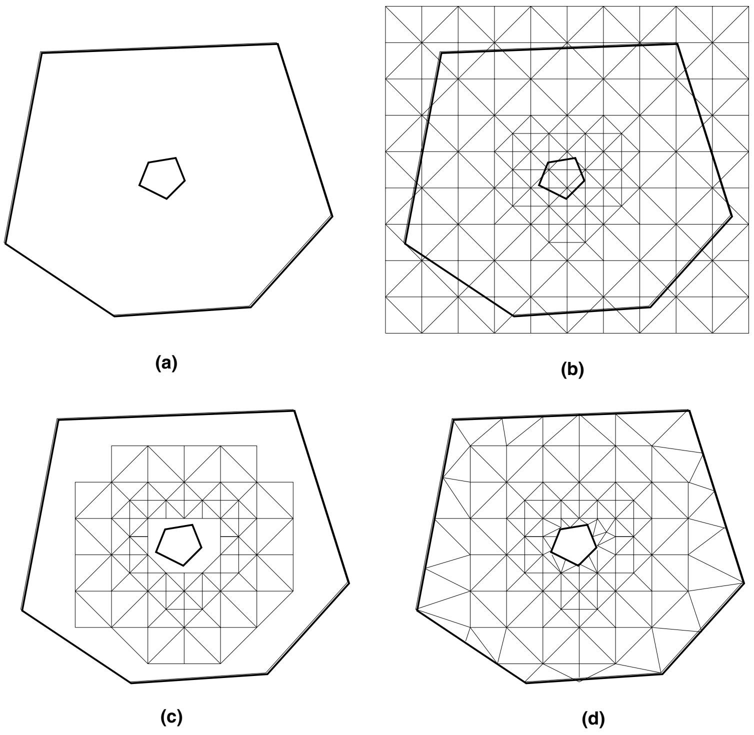


Figure 3.38. Mesh generation with adaptive cartesian core: (a) initial surface discretization; (b) Cartesian (adaptive) grid; (c) retain valid part; (d) complete mesh.

of faces, which, together with the surface discretization, form the initial front of the as yet ungridded portion of the domain. One can then use any of the techniques described above to mesh this portion of the domain. The capability to mesh core regions with space-filling tetrahedra only requires a small change within an unstructured grid generator. The core meshing is done in an independent module whose main role is to supply an additional list of triangles for the initial front. Meshing the core region is extremely fast, so that overall CPU requirements for large grids decrease considerably. A preliminary version of such a procedure (without the adaptive refinement of the Cartesian core) was proposed by Darve and Löhner (1997). Other techniques that generate unstructured grids with Cartesian cores are all those that use point distributions from regular Cartesian grids (Baker (1987)) or octrees (Shepard and Georges (1991), Kallinderis and Ward (1992)).

3.11. Volume-to-surface meshing

With the advent of robust volume meshing techniques, the definition and discretization of surfaces has become a considerable bottleneck in many CFD applications. Going from a typical CAD file to a watertight surface description suitable for surface meshing is far from trivial. For the external aerodynamic analysis of a complete car, this step can take approximately a month, far too long to have any impact on the design process. Given that the surface mesh requires a ‘clean’, watertight surface definition, a possible alternative is to first cover the domain with a volume mesh, and then modify this volume mesh so that it aligns itself with the boundaries (see section 3.5(c) above for possible alternatives). Moving a point to the boundary (even if surface patches are overlapping) is perceived as a simple operation that may be automated, whereas cleaning the geometry requires considerable manual intervention. The volume grid may be refined as specified by the user or in order to resolve surface features (e.g. curvature). Two options have been pursued in order to obtain a body-conforming mesh:

(a) split the tetrahedra crossed by the surface, thus inserting smaller tetrahedra and additional points; and

(b) move nodes to the surface and smooth the mesh.

The first option can lead to very small tetrahedra, and has not seen widespread acceptance. We will therefore concentrate on the second approach, where nodes will have to be moved in order to conform the volume mesh to the given boundary. The neighbouring nodes may also have to be moved in order to obtain a smooth mesh. Therefore, as a first step, a set of vertices that will be pushed/moved towards the boundary must be chosen. One solution is to classify all tetrahedra as being inside, outside or intersecting the boundary of the model (Marroquim *et al.* (2005)). All exterior (or interior) nodes of the intersecting tetrahedra are then marked for movement. However, this simple choice does not prove sufficient. Some tetrahedra might have all four vertices on the boundary, and therefore their volume may vanish once the vertices are moved. Also, some nodes may be far from the boundary, creating highly distorted elements after the vertices are moved. Following Molino *et al.* (2003) and Marroquim *et al.* (2005), we denote as the d-mesh the portion of the original volume mesh whose boundary is going to be moved towards the true boundary. Ideally, the d-mesh should be as close as possible to the true boundary. This can be obtained by defining an enveloped set of points (see Figure 3.39). All tetrahedra touching any of these points are considered as part of the d-mesh. The enveloped set contains points with all incident edges at least 25% inside the model (other percentages can be tried, but this seems a near-optimal value). This guarantees that tetrahedra of the d-mesh have at least one point inside the domain of interest. All points of the d-mesh that do not belong to the enveloped set are marked for movement towards the true boundary. Note that this choice of d-mesh ensures that none of the tetrahedra has four points marked for movement (Marroquim *et al.* (2005)).

While very flexible, the volume-to-surface approach has problems dealing with sharp edges or any feature with a size less than the smallest size h present in the mesh. In many engineering geometries, sharp edges and corners play a considerable role. On the other hand, this shortcoming may also be used advantageously to automatically filter out any feature with size less than h. Many CAD files are full of these ‘subgrid’ features (nuts, bolts, crevices, etc.) that are not required for a simulation and would require hours of manual removal for surface meshing. Volume-to-surface techniques remove these small features automatically. This class of meshing techniques has been used mainly to generate grids from images or other remote sensing data, where, due to voxelization, no feature thinner than h is present, and, *de facto*, no sharp ridges exist.

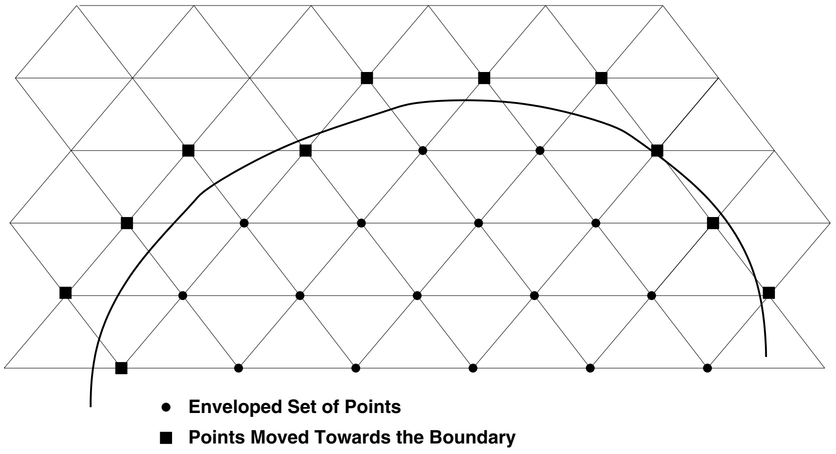

● Enveloped Set of Points
■ Points Moved Towards the Boundary

Figure 3.39. Enveloped set of points

3.12. Navier–Stokes gridding techniques

The need to grid complex geometries for the simulation of flows using the RANS equations, i.e. including the effects of viscosity and the associated boundary or mixing layers, is encountered commonly in engineering practice. The difficulty of this task increases not only with the geometric complexity of the domain to be gridded, but also with the Reynolds number of the flow. For high Reynolds numbers, the proper discretization of the very thin, yet important boundary or mixing layers requires elements with aspect ratios well in excess of 1:1,000. This requirement presents formidable difficulties to general, ‘black-box’ unstructured grid generators. These difficulties can be grouped into two main categories.

(a) *Amount of manual input.* As described above, most general unstructured grid generators employ some form of background grid or sources to input the desired spatial distribution of element size and shape. This seems natural within an adaptive context, as a given grid, combined with a suitable error indicator/estimator, can then be used as a background grid to generate an even better grid for the problem at hand. Consider now trying to generate from manual input a first grid that achieves stretching ratios in excess of 1:1000. The number of background gridpoints or sources required will be proportional to the curvature of the objects immersed in the flowfield. This implies an enormous amount of manual labour for general geometries, rendering this approach impractical.

(b) *Loss of control.* Most unstructured grid generators introduce a point or element at a time, checking the surrounding neighbourhood for compatibility. These checks involve Jacobians of elements and their inverses, distance functions and other geometrical operations that involve multiple products of coordinate differences. It is not difficult to see that as the stretching-ratio increases, round-off errors can become a problem. For a domain spanning 1000 m (the mesh around a Boeing-747), with a minimum element length at the wing of less than 0.01 mm across the boundary layer and 0.05 m along the boundary layer and along the wing, and a maximum element length of 20 m in the far-field, the ratio of element volumes is of the order of 3×10^{-12}. Although this is well within reach of the 10^{-16} accuracy of 64-bit arithmetic, element distortion and surface singularities, as well as loss of control of element shape, can quickly push this ratio to the limit.

Given these difficulties, it is not surprising that, at present, a ‘black-box’ unstructured (or structured, for that matter) grid generator that can produce acceptable meshes with such high aspect ratio elements does not exist. With the demand for RANS calculations in or past complex geometries being great, a number of semi-automatic grid generators have been devised. The most common way to generate meshes suitable for RANS calculations for complex geometries is to employ a structured or semi-structured mesh close to wetted surfaces or wakes. This ‘Navier–Stokes’ region mesh is then linked to an outer unstructured grid that covers the ‘inviscid’ regions. In this way, the geometric complexity is solved using unstructured grids and the physical complexity of near-wall or wake regions is solved by semi-structured grids. This approach has proven very powerful in the past, as evidenced by many examples.

The meshes in the semi-structured region can be constructed to be either quads/bricks (Nakahashi (1987) Nakahashi and Obayashi (1987)) or triangles/prisms (Kallinderis and Ward (1992), Löhner (1993), Pirzadeh (1993b, 1994)). The prisms can then be subdivided into tetrahedra if so desired. For the inviscid (unstructured) regions, multiblock approaches have been used for quads/bricks, and advancing front, Voronoi and modified quadtree/octree approaches for triangles/tetrahedra.

A recurring problem in all of these approaches has been how to link the semi-structured mesh region with the unstructured mesh region. Some solutions put forward have been considered.

- *Overlapped structured grids*. These are the so-called chimera grids (Benek (1985), Meakin and Suhs (1989), Dougherty and Kuan (1989)) that have become popular for RANS calculations of complex geometries; as the gridpoints of the various meshes do not coincide, they allow great flexibility and are easy to construct, but the solution has to be interpolated between grids, which may lead to higher CPU costs and a deterioration in solution quality.

- *Overlapped structured/unstructured grids*. In this case the overlap zone can be restricted to one cell, with the points coinciding exactly, so that there are no interpolation problems (Nakahashi (1987) Nakahashi and Obayashi (1987)).

- *Delaunay triangulation of points generated by algebraic grids*. In this case several structured grids are generated, and their spatial mapping functions are stored; the resulting cloud of points is then gridded using DTTs (Mavriplis (1990, 1991a)).

Although some practical problems have been solved by these approaches, they cannot be considered general, as they suffer from the following constraints.

- The first two approaches require a very close link between solver, grid generator and interpolation techniques to achieve good results; from the standpoint of generality, such a close link between solver, grid generator and interpolation modules is undesirable.

- Another problem associated with the first two approaches is that, at concave corners, negative (i.e. folded) or badly shaped elements may be generated. The usual recourse is to smooth the mesh repeatedly or use some other device to introduce ellipticity (Nakahashi (1988), Kallinderis and Ward (1992)). These approaches tend to be CPU intensive and require considerable expertise from the user. Therefore, they cannot be considered general approaches.

- The third case requires a library of algebraic grids to mesh individual cases, and can therefore not be considered a general tool. However, it has been used extensively for important specialized applications, e.g. single or multi-element airfoil flows (Mavriplis (1990)).

As can be seen, most of these approaches lack generality. Moreover, while they work well for the special application they were developed for, they are bound to be ineffective for others. The present section will describe one possibility for a general Navier–Stokes gridding tool. Strategies that are similar have been put forward by Pirzadeh (1993b, 1994), Müller (1993) and Morgan *et al.* (1993). Most of the element removal criteria, the smoothing of normals and the choice of point distributions normal to walls are common to all of these approaches.

The aim is to arrive at a single unstructured mesh consisting of triangles or tetrahedra that is suitable for Navier–Stokes applications. This mesh can then be considered completely independent of flow solvers, and neither requires any interpolation or other transfer operators between grids, nor the storage of mapping functions.

3.12.1. DESIGN CRITERIA FOR RANS GRIDDERS

Desirable design criteria for RANS gridders are as follows.

- The geometric flexibility of the unstructured grid generator should not be compromised for RANS meshes. This implies using unstructured grids for the surface discretization.

- The manual input required for a desired RANS mesh should be as low as that used for the Euler case. This requirement may be met by specifying at the points of the background grid the boundary layer thickness and the geometric progression normal to the surface.

- The generation of the semi-structured grid should be fast. Experience shows that usually more than half of the elements of a typical RANS mesh are located in the boundary-layer regions. This requirement can be met by constructing the semi-structured grids with the same normals as encountered on the surface, i.e. without recurring to smoothing procedures as the semi-structured mesh is advanced into the field (Nakahashi (1988), Kallinderis and Ward (1992)).

- The element size and shape should vary smoothly when going from the semi-structured to the fully unstructured mesh regions. How to accomplish this is detailed in subsequent sections.

- The grid generation procedure should avoid all of the problems typically associated with the generation of RANS meshes for regions with high surface curvature: negative or deformed elements due to converging normals, and elements that get too large due to diverging normals at the surface. In order to circumvent these problems, the same techniques which are used to achieve a smooth matching of semi-structured and unstructured mesh regions are used.

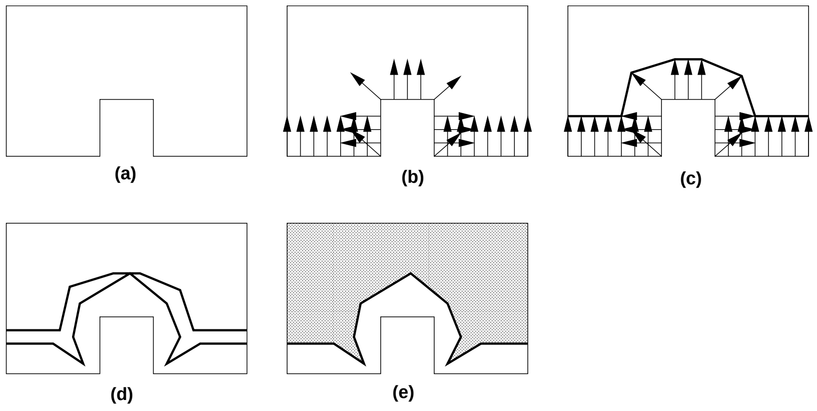


Figure 3.40. Generation of grids suitable for Navier–Stokes problems. (a) Define surface; (b) compute surface normals; (c) obtain boundary layer mesh; (d) remove bad elements; (e) complete unstructured mesh.

Given these design criteria, as well as the approaches used to meet them, this RANS grid generation algorithm can be summarized as follows (see Figure 3.40).

M1. Given a surface definition and a background grid, generate a surface triangulation using an unstructured grid generator.

M2. From the surface triangulation, obtain the surface normals.

M3. Smooth the surface normals in order to obtain a more uniform mesh in regions with high surface curvature.

M4. Construct a semi-structured grid with the information provided by the background grid and the smoothed normals.

M5. Examine each element in this semi-structured region for size and shape; remove all elements that do not meet certain specified quality criteria.

M6. Examine whether elements in this semi-structured region cross each other; if so, keep the smaller elements and remove the larger ones, until no crossing occurs.

M7. Examine whether elements in this semi-structured region cross boundaries; if so, remove the crossing elements.

M8. Mesh the as yet ‘empty’ regions of the computational domain using an unstructured grid generator in combination with the desired element size and shape.

Strategies that are similar to the one outlined above have been put forward by Pirzadeh (1993b, 1994), Müller (1993) and Morgan *et al.* (1993). The one closest to the algorithm described above is the advancing layers method of Pirzadeh (1993b, 1994), which can be obtained by repeatedly performing steps M4 to M7, one layer at a time. At the time the present technique was conceived, it was felt that generating and checking a large number of elements at the same time would offer vectorization benefits. Regardless of the approach used, most of the algorithmic techniques described in the following

- element removal criteria;
- smoothing of normals;
- choice of point distributions normal to walls;
- subdivision of prisms into tetrahedra;
- speeding up testing and search, etc.

are common and can be used for all of them.

3.12.2. SMOOTHING OF SURFACE NORMALS

Smoothing of surface normals is always advisable for regions with high surface curvature, particularly corners, ridges and intersections. The basic smoothing step consists of a pass over the faces in which the following operations are performed:

```
Given:
    - The point-normals rnor0
    - The boundary conditions for point-normals
Then:
    - Initialize new point-normals rnor1=0
    - do: loop over the faces:
        - Obtain the points of this face;
        - Compute average face normal rnofa from rnor0;
        - Add rnofa to rnor1;
    - enddo
    - Normalize rnor1;
    - Apply boundary conditions to rnor1
```

In order to start the smoothing process, initial point-normals `rnor0`, as well as boundary conditions for point-normals, must be provided. In particular, the choice of boundary conditions is crucial in order to ensure that no negative elements are produced at corners, ridges and intersections. Figure 3.41 shows a number of possibilities.

Note that the trailing edge of wings (a particularly difficult case if due care is not taken) falls under one of these categories. In order to obtain proper boundary conditions, a first pass is made over all of the faces (wetted and non-wetted), computing the face-normals `rnofa`. In a second pass, the average surface normal that results at each point from its surrounding wetted faces is compared to the face-normals. If too large a discrepancy between a particular wetted face-normal and the corresponding point-normal is detected, the point is marked as being subjected to boundary conditions. For the discrepancy, a simple scalar-product-of-normals test is employed. For each of these special points, the number of distinct

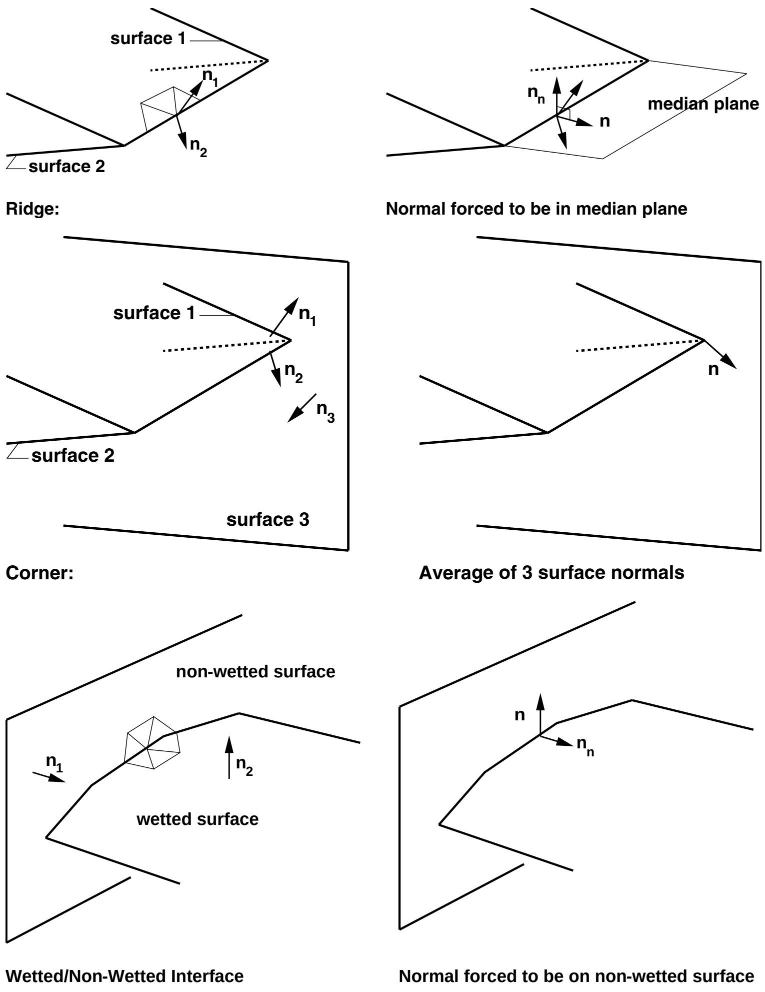


Figure 3.41. Boundary conditions for the smoothing of normals

‘surfaces’ is evaluated (again with scalar product tests). If two distinct ‘surfaces’ are detected, a ridge boundary condition is imposed, i.e. the normal of that given point must lie in the plane determined by the average normal of the two ‘surfaces’ (see Figure 3.41(a)). If three or more distinct ‘surfaces’ are detected, a corner boundary condition is imposed, i.e. the normal of that given point is given by the average of the individual ‘surface’ normals (see Figure 3.41(b)). For the points lying on the wetted/non-wetted interface, the normals are imposed to be in the plane given by the averaged point-normal of the non-wetted faces (see Figure 3.41(c)).

Given the boundary conditions for the normals, the initial point-normals are obtained by taking the point-averaged normals from the wetted surface normals and applying the boundary conditions to them.

The complete smoothing procedure, applied as described above, may require in excess of 200 passes in order to converge. This slow convergence may be speeded up considerably through the use of conjugate gradient (Hestenes and Stiefel (1952)) or superstep (Gentzsch and Schlüter (1978), Löhner and Morgan (1987)) acceleration procedures. Employing the latter procedures, convergence is usually achieved in less than 20 passes.

3.12.3. POINT DISTRIBUTION ALONG NORMALS

Ideally, we would prefer to have normals that are perpendicular to the surface in the immediate vicinity of the body and smoothed normals away from the surface (Weatherill *et al.* (1993a)). Such a blend may be accomplished by using Hermitian polynomials. If we assume given:

- the surface points $\mathbf{x}_0$,
- the boundary layer thickness δ,
- the surface normals $\mathbf{n}_0, \mathbf{n}_1$ before and after smoothing,
- a non-dimensional boundary layer point-distribution parameter ξ of the form $\xi_{i+1} = \alpha\xi_i, \xi_n = 1$,

then the following Hermitian cubic polynomial in ξ will produce the desired effect:

$$\mathbf{x} = \mathbf{x}_0 + \xi\delta\mathbf{n}_0 + \xi \cdot (2 - \xi) \cdot \xi\delta(\mathbf{n}_1 - \mathbf{n}_0). \tag{3.39}$$

One can readily identify the linear parts $\xi\delta\mathbf{n}_i$. In some cases, a more pronounced (and beneficial) effect may be produced by substituting for the higher-order polynomials a new variable η,

$$\xi \cdot (2 - \xi) :\rightarrow \eta \cdot (2 - \eta), \quad \eta = \xi^p, \tag{3.40}$$

where, e.g., $p = 0.5$. The effect of using such a Hermitian polynomial to blend the normal and smoothed normal vectors is depicted in Figure 3.42. As may be seen, the smaller the values of p, the faster the normal tends to the smoothed normal $\mathbf{n}_1$.

3.12.4. SUBDIVISION OF PRISMS INTO TETRAHEDRA

In order to form tetrahedra, the prisms obtained by extruding the surface triangles along the smoothed normals must be subdivided. This subdivision must be performed in such a way that the diagonals introduced at the rectangular faces of the prisms match across prisms. The shortest of the two possible diagonals of each of these rectangular faces is chosen in order to avoid large internal angles, as shown in Figure 3.43.

As this choice only depends on the coordinates and normals of the two endpoints of a surface side, compatibility across faces is assured. The problem is, however, that a prism cannot be subdivided into tetrahedra in an arbitrary way. Therefore, care has to be taken when choosing these diagonals. Figure 3.44 illustrates the possible diagonals as the base sides of the prism are traversed.

One can see that in order to obtain a combination of diagonals that leads to a possible subdivision of prisms into tetrahedra, the sides of the triangular can not be up-down or down-up as one traverses them. This implies that the sides of the triangular base mesh have to be

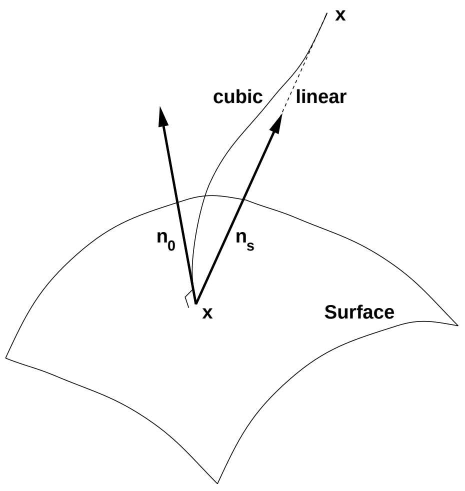


Figure 3.42. Blending of smoothed and unsmoothed surface normals

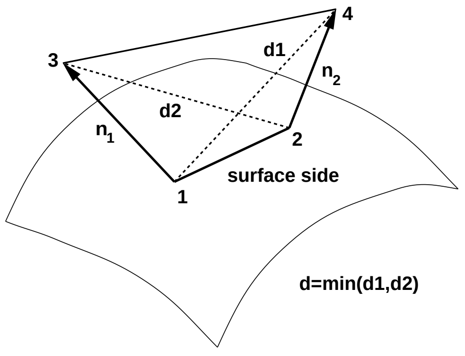


Figure 3.43. Choice of diagonal for prism faces

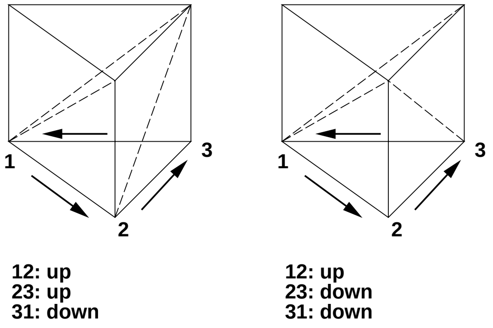


Figure 3.44. Possible subdivision patterns for prisms

marked in such a way that no such combination occurs. The following iterative procedure can be used to arrive at valid side combinations:

```
D0. Given:
- The sides of the surface triangulation;
- The sides of each surface triangle;
- The triangles that surround each surface triangle;
D1. Initialize ipass=0;
D2. Initialize lface(1:nface)=0;
D3. do iface=1,nface: loop over the surface triangles
D4. if(lface(iface).eq.0) then
- if the current side combination is not valid:
       - do: loop over the sides of the triangle
          - if(ipass.eq.0) then
              - if: the inversion of the side/diagonal orientation
                leads to an allowed side combination
                in the neighbour-triangle jface:
                - Invert the side/diagonal orientation
                - lface(iface)=1
                - lface(jface)=1
                - Goto next face
              endif
          - else
                - Invert the side/diagonal orientation
                - lface(iface)=1
                - lface(jface)=1
          endif
          endif
       enddo
   endif
enddo
D5. if: any unallowed combinations are left:
  ipass=ipass+1
  goto D2
endif
```

When inverting the side/diagonal orientation, those diagonals with the smallest maximum internal angle are sampled first in order to minimize the appearance of bad elements. The procedure outlined above works well, converging in at most two passes over the surface mesh for all cases tested to date. Moreover, even for large surface grids ($>$50 000 points), the number of diagonals that require inversion of side/diagonal orientation is very small ($<$15).

3.12.5. ELEMENT REMOVAL CRITERIA

The critical element of any matching algorithm is the development of good element removal criteria. The criteria to be considered are: element size, element shape, element overlap and element crossing of boundary faces.

3.12.5.1. Element size

The two main types of problems encountered in semi-structured grid regions that are related to element size are elements that are either too large or negative (folded). These problems originate for different reasons, and are therefore best treated separately.

Large elements

As a result of surface normals diverging close to convex surfaces very large elements (as compared to the user-defined size and shape) may appear in the semi-structured mesh regions. The situation is shown diagrammatically in Figure 3.45.

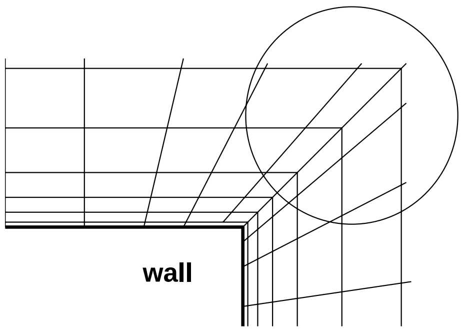


Figure 3.45. Removal of large elements

The volume of each element in the semi-structured mesh region is compared to the element volume desired by the user for the particular location in space. Any element with a volume greater than the one specified by the user (e.g. via background grids and sources) is marked for deletion. Tree-search algorithms (see Chapter 2) are used to relate the information between the background grid and a particular location in space.

Negative elements

As a result of folding away from concave surfaces, elements with negative Jacobians may appear. The situation is shown diagrammatically in Figure 3.46. As before, the element volumes are computed. All elements with negative volumes are marked for deletion.

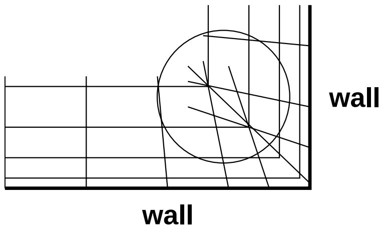


Figure 3.46. Removal of negative (folded) elements

An observation often made for complex geometries is that, typically, the elements adjacent to negative elements tend to be highly deformed. Therefore, one may also remove all elements that have points in common with negative elements. This one-pass procedure can be extended to several passes, i.e. neighbours of neighbours, etc. Experience indicates, however, that one pass is sufficient for most cases.

3.12.5.2. Element shape

The aim of a semi-structured mesh close to a wall is to provide elements with very small size normal to the wall and reasonable size along the wall. Due to different meshing requirements along the wall (e.g. corners, separation points, leading and trailing edges for small element size, other regions with larger element size), as well as the stated desire for a low amount of user input, elements that are longer in the direction normal to the wall than along the wall may appear. The situation is sketched diagrammatically in Figure 3.47.

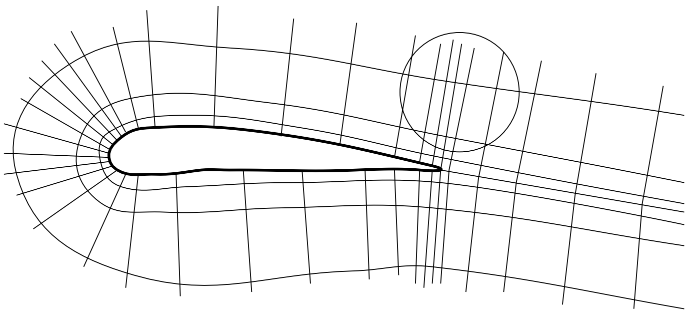


Figure 3.47. Removal of badly shaped elements

For the semi-structured grids, the element and point numbering can be assumed known. Therefore, a local element analysis can be performed to determine whether the element has side ratios that are consistent with boundary layer gridding. All elements that do not satisfy this criterion, i.e. that are longer in the direction normal to the wall than along the wall, are marked for deletion.

3.12.5.3. Overlapping elements

Overlapping elements will occur in regions close to concave surfaces with high curvature or when the semi-structured grids of two close objects overlap. Another possible scenario is the overlap of the semi-structured grids of mixing wakes. The main criterion employed is to keep the smaller element whenever an overlap occurs. In this way, the small elements close to surfaces are always retained. Straightforward testing would result in $O(N_{el})$ operations per element, where N_{el} denotes the number of elements, leading to a total number of operations of $O(N_{el}^2)$. By using quad/octrees or any of the other suitable data structures discussed in Chapter 2, the number of elements tested can be reduced significantly, leading to a total number of operations of $O(N_{el} \log N_{el})$.

For quad/octrees, the complete testing procedure looks like the following:

```
- Construct a quad/octree for the points;
- Order the elements according to decreasing volume (e.g. in a heap-list);
- Construct a linked list for all the elements surrounding each point;
- do: Loop over the elements, in descending volume, testing:
- if the element, denoted in the following by ielem, has not been marked for
   deletion before:
    - Obtain the minimum/maximum extent of the coordinates belonging to this element;
    - Find from the quad/octree all points falling into this search region, storing them in a list
      lclop(1:nclop);
    - Find all the unmarked elements with smaller volume than ielem surrounding the
      points stored in
      lclop(1:nclop); this yields a list of close elements lcloe(1:ncloe);
    - Loop over the elements stored in lcloe(1:ncloe):
      -if the element crosses the faces of or is inside ielem
           Mark ielem for deletion
       endif
    endif
    enddo
```

The reason for looping over the elements according to descending volumes is that the search region is obtained in a natural way (the extent of the element). Looping according to ascending volumes would imply guessing search regions. As negative elements could lead to a failure of this test, the overlap test is performed after the negative elements have been identified and marked.

The test for overlapping elements can account for a large portion of the overall CPU requirement. Therefore, several filtering and marking strategies have to be implemented to speed up the procedure. The most important ones are as follows.

Column marking

The mesh crossing test is carried out after all the negative, badly shaped and large elements have been removed. This leaves a series of prismatic columns, which go from the surface triangle to the last element of the original column still kept. The idea is to test the overlap of these prismatic columns, and mark all the elements in columns that pass the test as not requiring any further crossing tests. In order to perform this test, we subdivide the prismatic columns into three tetrahedra as before, and use the usual element crossing tests for tetrahedra. Since a large portion of the elements does not require further testing (e.g. convex surfaces that are far enough apart), this is a very effective test that leads to a drastic reduction of CPU requirements.

Prism removal

Given that the elements in the semi-structured region were created from prisms, it is an easy matter to identify for each element the triplet of elements stemming from the same prism. If an element happens to be rejected, all three elements of the original prism are rejected. This avoids subsequent testing of the elements from the same prism.

Marking of surfaces

Typically, the elements of a surface segment or patch will not overlap. This is because, in most instances, the number of faces created on each of these patches is relatively large, and/or the patches themselves are relatively smooth surfaces. The main areas where overlap can occur are corners or ‘coalescing fronts’. For both cases, in the majority of the cases encountered in practice, the elements will originate from different surface patches. Should a patch definition of the surface not be available, an alternative is to compute the surface smoothness and concavity from the surface triangulation. Then discrete ‘patches’ can be associated with the discretized surface using, e.g., a neighbour-to-neighbour marching algorithm. If the assumption of surface patch smoothness and/or convexity can be made, then it is clear that only the elements (and points) that originated from a surface patch other than the one that gave rise to the element currently being examined need to be tested. In this way, a large number of unnecessary tests can be avoided.

Rejection via close points

The idea of storing the patch from which an element emanated can also be applied to points. If any given point from another patch that is surrounded by elements that are smaller than the one currently being tested is too close, the element is marked for deletion. The proximity test is carried out by computing the smallest distance between the close point and the four vertices of the element being tested. If this distance is less than approximately the smallest side length of the element, the element is marked for deletion. Evaluating four distances is very inexpensive compared to a full crossing test.

Rejection if a point lies within an element

If one of the close points happens to fall within the element being tested, then obviously a crossing situation occurs. Testing whether a point falls inside the element is considerably cheaper (by more than an order of magnitude) than testing whether the elements surrounding this point cross the element. Therefore, all the close points are subjected to this test before proceeding.

Top element in prism test

The most likely candidate for element crossing of any given triplet of elements that form a prism is the top one. This is because, in the case of ‘coalescing fronts’, the top elements will be the first ones to collide. It is therefore prudent to subject only this element to the full (and expensive) element crossing test. In fact, only the top face of this element needs to be tested. This avoids a very large number of unnecessary tests, and has been found to work very well in practice.

Avoidance of low layer testing

Grids suitable for RANS calculations are characterized by having extremely small grid spacings close to wetted surfaces. It is highly unlikely – although of course not impossible – that the layers closest to the body should cross or overlap. Therefore, one can, in most instances, avoid the element crossing test for the elements closest to wetted surfaces. For typical grids, between four and ten layers in the grids are avoided. This number is, however, problem dependent, but fairly easy to estimate if a graphical workstation is at hand. For most applications, such a device is nowadays commonly used to prepare the data.

Rejection via faces of an element

After all the filtering operations described above, a considerable number of elements will still have to be subjected to the full crossing test. The crossing test is carried out face-by-face, using the face-crossing check described in Section 3.6.1. If all the points of a close element lie completely on the outward side of any of the faces of the element being tested, this pair of elements cannot possibly cross each other. This filter only involves a few scalar products, and is very effective in pruning the list of close elements tested.

3.12.5.4. *Elements crossing boundary faces*

In regions where the distance between surfaces is very small, elements from the semi-structured region are likely to cross boundary faces. As this test is performed after the element crossing tests are conducted, the only boundaries which need to be treated are those that have no semi-structured grid attached to them. In order to detect whether overlapping occurs, a loop is performed over the surface faces, seeing if any element crosses it. As before, straightforward testing would result in an expensive $O(N_{el} \cdot N_f)$ procedure, where N_f denotes the number of boundary faces. By using quad/octrees, this complexity can be reduced to $O(N_f \log N_{el})$. The face-crossing check looks essentially the same as the check for overlapping elements, and its explicit description is therefore omitted. Needless to say, some of the filtering techniques described before can also be applied here.

3.12.5.5. *Distribution of points normal to wetted walls/wakes*

To gain some insight into the distribution of points required for typical high-Reynolds-number flows, let us consider the case of the flat plate. The friction coefficient c'_f, given by

$$c'_f = \frac{2\tau_w}{\rho v_\infty^2}, \tag{3.41}$$

is a well-documented quantity. Here τ_w denotes the shear at the wall, ρ a reference density and v_∞ a reference velocity. The shear in the laminar sublayer is given by

$$\tau_w = \mu u_{,y}, \tag{3.42}$$

where u denotes the velocity parallel to the wall and y the direction normal to it. Assuming that

$$c'_f = \alpha Re_x^\beta, \tag{3.43}$$

where x denotes the streamwise location, we have

$$\tau_w = \mu u_{,y} = c'_f \frac{\rho v_\infty^2}{2}, \tag{3.44}$$

implying

$$u_{,y} = \frac{\alpha}{2}\frac{v_\infty}{x}Re_x^{1+\beta}. \tag{3.45}$$

In order to obtain a meaningful gradient of the velocity at the wall, the velocity of the first point away from the wall, u_1, should only be a fraction ϵ of the reference velocity v_∞. Therefore,

$$u_{,y} \approx \frac{u_1 - u_0}{\Delta y} = \frac{\epsilon v_\infty}{\Delta y} = \frac{\alpha}{2}\frac{v_\infty}{x}Re_x^{1+\beta}, \tag{3.46}$$

which yields the required height of the first point above the wall:

$$\Delta y = \frac{2\epsilon x}{\alpha Re_x^{1+\beta}}. \tag{3.47}$$

For *laminar* flow, we have (Schlichting (1979)) $\alpha = 0.664$, $\beta = -1/2$, implying that

$$\Delta y = 3\epsilon x Re_x^{-1/2}, \tag{3.48}$$

while for turbulent flow $\alpha = 0.045$, $\beta = -1/4$, which yields

$$\Delta y = 44\epsilon x Re_x^{-3/4}. \tag{3.49}$$

Typical values for ϵ range from $\epsilon = 0.05$ for laminar flow to $\epsilon = 0.01$ for turbulent flows. For turbulent flows, an alternative way of obtaining the location of the first point away from the wall can be obtained via the non-dimensional distance parameter y^+, defined as

$$y^+ = \frac{y}{\mu}\sqrt{\rho\tau_w}. \tag{3.50}$$

Given y^+ for the first point away from the wall, one can determine Δy from (3.45):

$$\Delta y = \frac{y^+\mu}{\sqrt{\rho\tau_w}} = \frac{y^+\mu}{\sqrt{\rho\mu u_{,y}}} = \frac{y^+ x}{\sqrt{\alpha/2}Re_x^{1+\beta/2}}. \tag{3.51}$$

For $\alpha = 0.045$, $\beta = -1/4$, this results in

$$\Delta y = 6.7 y^+ x Re_x^{-7/8}. \tag{3.52}$$

If the distance between points increases geometrically as

$$\Delta y_{i+1} = c\Delta y_i, \tag{3.53}$$

one can estimate the number of layers required to assure a smooth transition to the isotropic mesh outside the boundary layer region. Denoting by h_i the element size in the isotropic mesh region, the number of layers n required is given by

$$n = \frac{\log h_i - \log \Delta y_0}{\log c}, \tag{3.54}$$

and the total thickness of the semi-structured grid region is

$$\delta = \frac{c^{n+1} - 1}{c - 1}\Delta y_0. \tag{3.55}$$

Typical values for c are $1.10 < c < 1.30$.

3.13. Filling space with points/arbitrary objects

Many simulation techniques in computational mechanics require a space-filling cloud of arbitrary objects. For the case of ‘gridless’ or ‘mesh free’ PDE solvers (see Nay and Utku (1972), Gingold and Monahghan (1983), Batina (1993), Belytschko *et al.* (1994), Duarte and Oden (1995), Oñate *et al.* (1996a,b), Liu *et al.* (1996), Löhner *et al.* (2002)) these are simply points. For discrete element methods (see Cundall and Stack (1979), Cundall (1988), Cleary (1998), Sakaguchi and Murakami (2002)), these could be spheres, ellipsoids, polyhedra or any other arbitrary shape. The task is therefore to fill a prescribed volume with these objects so that they are close but do not overlap in an automatic way.

Several techniques have been used to place these objects in space. The so-called ‘fill and expand’ or ‘popcorn’ technique (Sakaguchi and Murakami (2002)) starts by first generating a coarse mesh for the volume to be filled. This subdivision of the volume into large, simple polyhedra (elements) is, in most cases, performed with hexahedra. The objects required (points, spheres, ellipsoids, polyhedra, etc.) are then placed randomly in each of these elements. These are then expanded in size until contact occurs or the desired fill ratio has been achieved. An obvious drawback of this technique is the requirement of a mesh generator to initiate the process. A second class of techniques are the ‘advancing front’ or ‘depositional’ methods (Löhner and Oñate (1998), Feng *et al.* (2002)). Starting from the surface, objects are added where empty space still exists. In contrast to the ‘fill and expand’ procedures, the objects are packed as close as required during introduction. Depending on how the objects are introduced, one can mimic gravitational or magnetic deposition, layer growing or size-based growth. Furthermore, so-called radius growing can be achieved by first generating a coarse cloud of objects, and then growing more objects around each of these. In this way, one can simulate granules or stone.

In the present section, we consider the direct generation of clouds of arbitrary objects with the same degree of flexibility as advanced unstructured mesh generators. The mean distance between objects (or, equivalently, the material density) is specified by means of background grids, sources and density attached to CAD entities. In order not to generate objects outside the computational domain, an initial triangulation of the surface that is compatible with the desired mean distance between objects specified by the user is generated. Starting from this initial ‘front’ of objects, new objects are added until no further objects can be introduced. Whereas the AFT for the generation of volume grids removes one face at a time to generate elements, the present scheme removes one object at a time, attempting to introduce as many objects as possible in its immediate neighbourhood.

3.13.1. THE ADVANCING FRONT SPACE-FILLING ALGORITHM

Assume as given:

- A specification of the desired mean distance between objects in space, as well as the mean size of these objects. This can be done through a combination of background grids, sources and mean distance to neighbours attached to CAD-data (see section 3.2 above).
- An initial triangulation of the surface, with the face normals pointing towards the interior of the domain to be filled with points.

With reference to Figure 3.48, which shows the filling of a simple 2-D domain with ellipsoids, the complete advancing front space-filling algorithm may be summarized as follows:

```
- Determine the required mean point distance for the points of the triangulation;
- while: there are active objects in the front:
   - Remove the object ioout with the smallest specified mean distance to
     neighbours from the front;
   - With the specified mean object distance: determine the coordinates of nposs
     possible new neighbours. This is done using a stencil, examples of which are shown
     in Figure 3.49;
   - Find all existing objects in the neighbourhood of ioout;
   - Find all boundary faces in the neighbourhood of ioout;
   - do: For each one of the possible new neighbour objects ionew:
     - If there exists an object closer than a minimum distance dminp
     from ionew, or if the objects are penetrating each other:
     ⇒ skip ionew;
   - If the object ionew crosses existing faces:
     ⇒ skip ionew;
   - If the line connecting ioout and ionew crosses existing faces:
     ⇒ skip ionew;
   - Determine the required mean point distance for ionew;
   - Increment the number of objects by one;
   - Introduce ionew to the list of coordinates and store its attributes;
   - Introduce ionew to the list of active front objects;
   enddo
endwhile
```

The main search operations required are:

- finding the active object with the smallest mean distance to neighbours;
- finding the existing objects in the neighbourhood of `ioout`;
- finding the boundary faces in the neighbourhood of `ioout`.

These three search operations are performed efficiently using heap lists, octrees and linked lists, respectively.

3.13.2. POINT/OBJECT PLACEMENT STENCILS

A number of different stencils may be contemplated. Each one of these corresponds to a particular space-filling point/object configuration. The simplest possible stencil is the one that only considers the six nearest-neighbours on a Cartesian grid (see Figure 3.49(a)). It is easy to see that this stencil, when applied recursively with the present advancing front algorithm, will fill a 3-D volume completely. Other Cartesian stencils, which include nearest-neighbours with distances $\sqrt{2}$ and $\sqrt{3}$ from `ioout`, are shown in Figures 3.49(b) and (c). The ‘tetrahedral’ stencil shown in Figure 3.49(d) represents another possibility. Furthermore, one can contemplate the use of random stencils, i.e. the use of n randomly selected directions

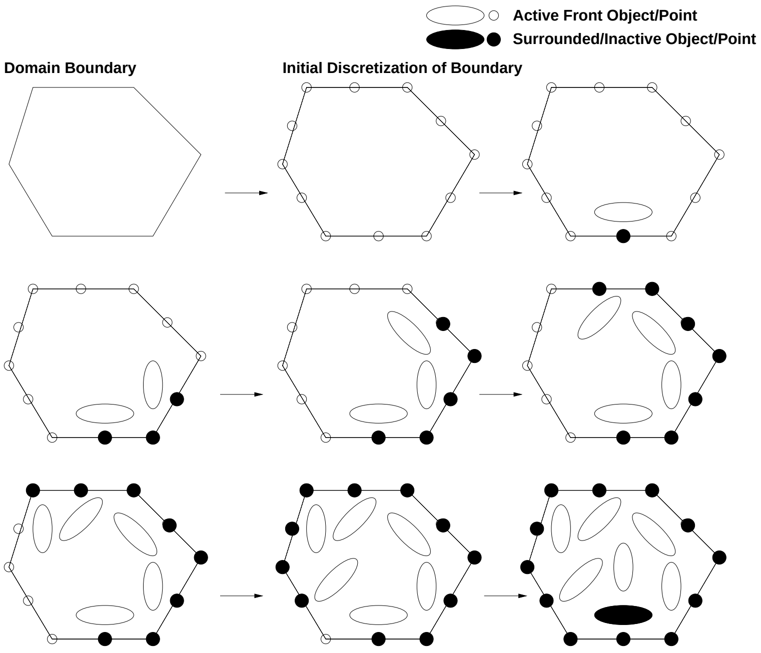

Active Front Object/Point
Surrounded/Inactive Object/Point
Domain Boundary
Initial Discretization of Boundary

Figure 3.48. Advancing front space-filling with ellipses

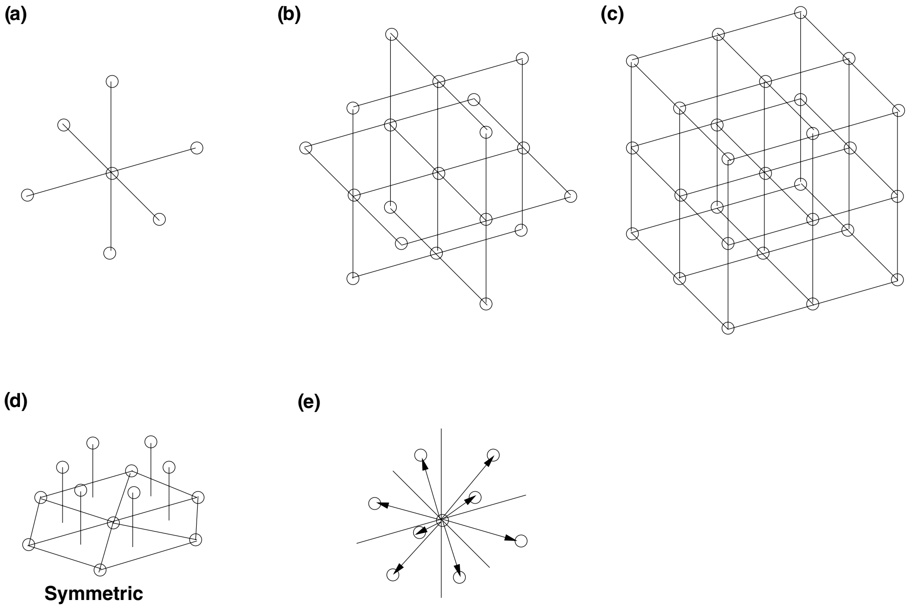


Figure 3.49. Point stencils: (a) Cartesian [6]; (b) Cartesian [18]; (c) Cartesian [26]; (d) tetrahedral [17]; (e) random.

to place new objects close to an existing one. For the generation of points and spheres, it was found that the six-point stencil leads to the smallest amount of rejections and unnecessary testing (Löhner and Oñate (1998)).

In many instances, it is advisable to generate 'body conforming' clouds of points in the vicinity of surfaces. In particular, Finite Point Methods (FPMs) or Smoothed Particle Hydrodynamics (SPH) techniques may require these 'boundary layer point distributions'. Such point distributions can be achieved by evaluating the average point-normals for the initial triangulation. When creating new points, the stencil used is rotated in the direction of the normal. The newly created points inherit the normal from the point `ioout` they originated from.

3.13.3. BOUNDARY CONSISTENCY CHECKS

A crucial requirement for a general space-filling object generator is the ability to only generate objects in the computational domain desired. Assuming that the object to be removed from the list of active objects `ionew` is inside the domain, a new object `ionew` will cross the boundary triangulation if it lies on the other side of the plane formed by any of the faces that are in the proximity of `ionew` and can see `ionew`. This criterion is made more stringent by introducing a tolerated closeness or tolerated distance d_t of new objects to the exterior faces of the domain. Testing for boundary consistency is then carried out using the following algorithm (see Figure 3.50):

```
- Obtain all the faces close to ioout;
- Filter, from this list, the faces that are pointing away from ioout;
- do: For each of the close faces:
    - Obtain the normal distance $d_n$ from ionew to this face;
            if: $d_n < 0$: ionew lies outside the domain
        $\Rightarrow$ reject ionew and exit;
    elseif: $d_n > d_t$: ionew if far enough from the faces
        $\Rightarrow$ proceed to the next close face;
    elseif: $0 \leq d_n \leq d_t$:
                    obtain the closest distance $d_{\min}$ of ionew to this face;
        if: $d_{\min} < d_t$: ionew is too close to the boundary
            $\Rightarrow$ reject ionew and exit;
     endif
- enddo
```

Typical values for d_t are $0.707d_0 \leq d_t \leq 0.9d_0$, where d_0 denotes the desired mean average distance between points.

3.13.4. MAXIMUM COMPACTION TECHNIQUES

For SPH and finite point applications, the use of stencils is sufficient to ensure a proper space filling (discretization) of the computational domain. However, many applications that consider not points but volume-occupying objects, such as spheres, ellipsoids and polyhedra, require a preset volume fraction occupied by these objects and, if possible, a minimum

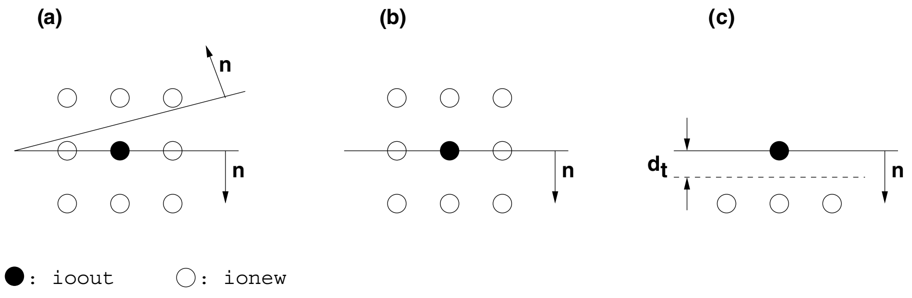


Figure 3.50. Boundary consistency checks: (a) obtain stencil for `ioout` and close faces; (b) filter faces pointing away from `ioout`; (c) retain points inside computational domain

number of contacts. The modelling of discontinua via discrete element methods represents a typical class of such applications. Experience indicates that the use of point stencils does not yield the desired volume fractions and contact neighbours. Two complementary techniques have proven useful to achieve these goals: closest object placement and move/enlarge post-processing.

3.13.4.1. Closest object placement

Closest object placement attempts to position new objects as close as possible to existing ones (see Figure 3.51). The initial position for new objects is taken from a stencil as before. The closest three objects to the new object position and the object being removed from the active front are then obtained. This does not represent any substantial increase in effort, as the existing objects in the vicinity are required anyhow for closeness/penetration testing. Starting from the three closest objects, an attempt is made to place the new object in such a way that it contacts all of them, does not penetrate any other objects and resides inside the computational domain. If this is not possible, an attempt is made with the two closest objects. Should this also prove impossible, an attempt is made to move the new objects towards the object being removed from the active front. If this attempt is also unsuccessful, the usual stencil position is chosen.

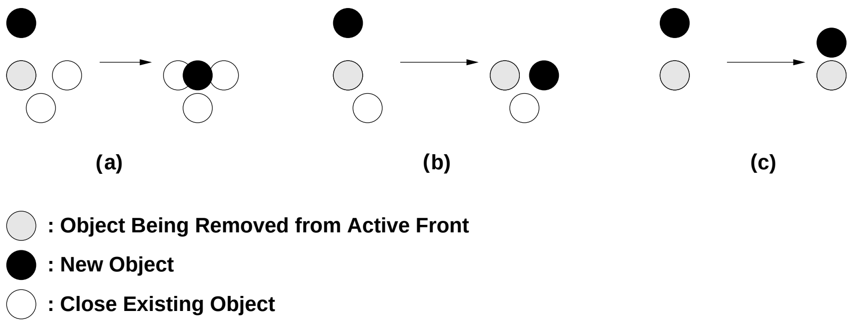


Figure 3.51. Closest object placement: (a) move to three neighbours; (b) move to two neighbours; (c) move to one neighbour

3.13.4.2. Move/enlarge post-processing

Whereas closest object placement is performed while space is being filled with objects, post-processing attempts to enlarge and/or move the objects in such a way that a higher volume ratio of objects is obtained, and more contacts with nearest-neighbours are established. The procedure, shown schematically in Figure 3.52, may be summarized as follows:

```
- while: objects can be moved/enlarged:
    - do: loop over the objects ioout:
    - Find the closest existing objects of ioout;
    - Find all boundary faces in the neighbourhood of ioout;
    - Move the object away from the closest existing objects so that:
      - The minimum distance to the closest existing objects increases;
      - ioout does not penetrate the boundary;
    - Enlarge object ioout by a small percentage
      - If the enlarged object ionew penetrates other objects or
        the boundary: revert to original size;
        ⇒ skip ioout;
      enddo
endwhile
```

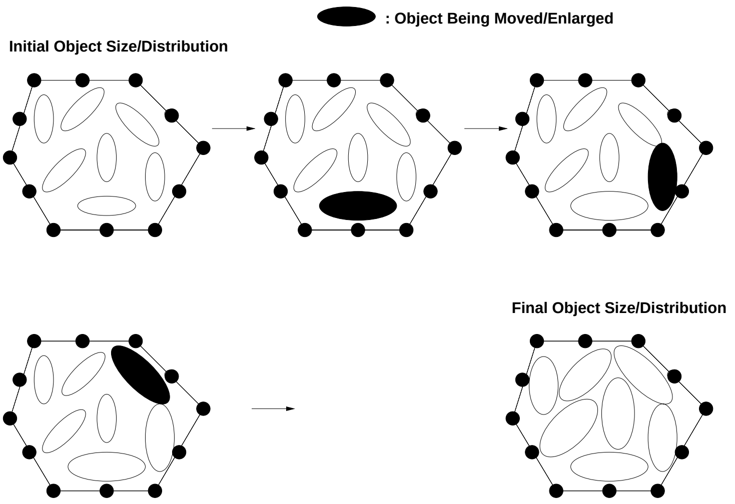

: Object Being Moved/Enlarged
Initial Object Size/Distribution
Final Object Size/Distribution

Figure 3.52. Movement and enlargement of objects

The increase factors are progressively decreased for each new loop over the objects. Typical initial increase ratios are 5%. As the movement of objects is moderate (e.g. less than the radius for spherical objects), the spatial search data structures (bins, octrees) required during space filling can be reused without modification. This move/enlarge post-processing is very fast and effective, and is used routinely for the generation of discrete element method/discrete particle method datasets.

3.13.5. ARBITRARY OBJECTS

The most time-consuming part of the present technique is given by the penetration/closeness checks. These checks are particularly simple for spheres. However, for arbitrary objects, the CPU burden can be significant. Specialized penetration/closeness checks are available for ellipsoids (Feng *et al.* (2002)), but for general polyhedra the faces have to be triangulated and detailed face/face checks are unavoidable. A recourse that has proven effective is to approximate arbitrary objects by a collection of spheres (see Figure 3.53). When adding a new object in space, the penetration/closeness checks are carried out for all spheres comprising the object. The new object is only added if all spheres pass the required tests (Löhner and Oñate (2004)).

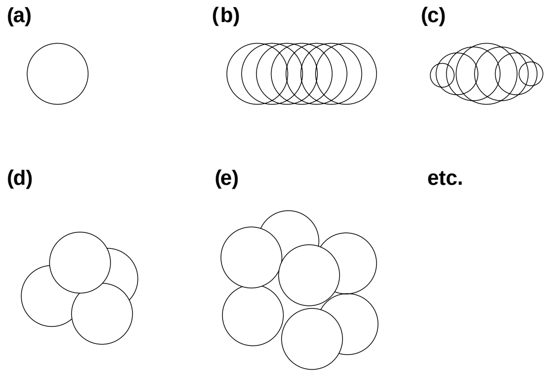


Figure 3.53. Arbitrary objects as a collection of spheres: (a) sphere; (b) tube; (c) ellipsoid; (d) tetrahedron; (e) cube

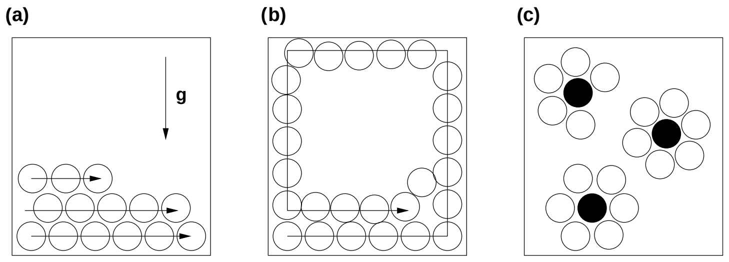
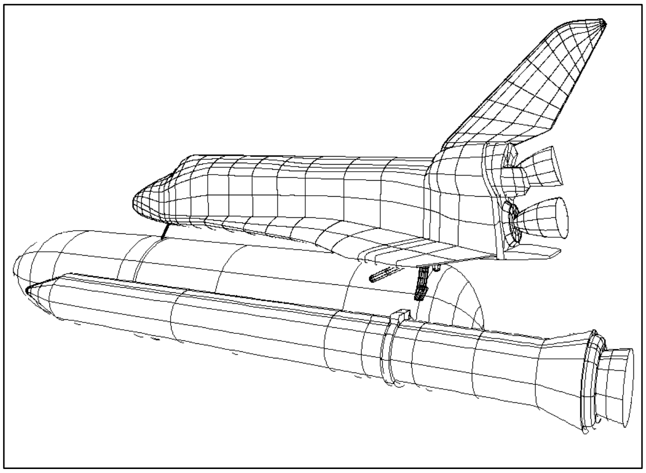


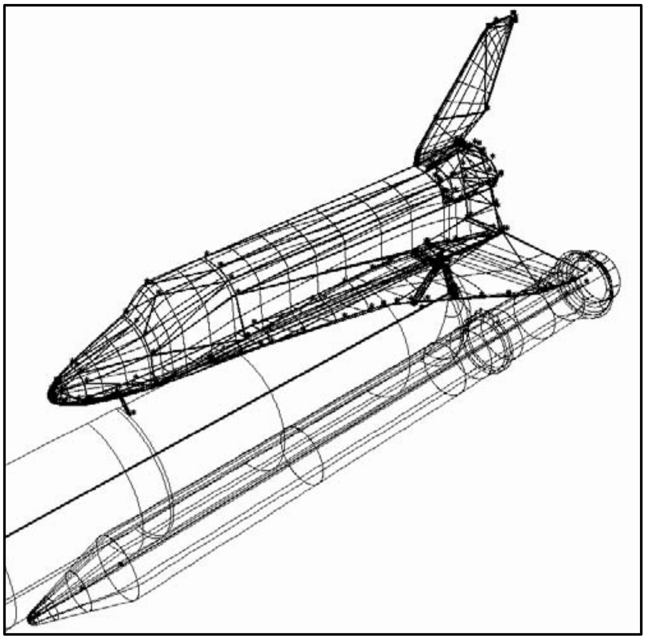
Figure 3.54. Deposition patterns: (a) force field; (b) layered growth; (c) seed-point (radius) growing

3.13.6. DEPOSITION PATTERNS

Depending on how the objects are removed from the active front, different deposition patterns can be achieved. The advancing front space-filling technique always removes the object with the smallest average distance (size) to new neighbours from the active front. Different size distributions will therefore lead to different deposition patterns (see Figure 3.54). Gravitational deposition can be achieved by specifying a size distribution that decreases



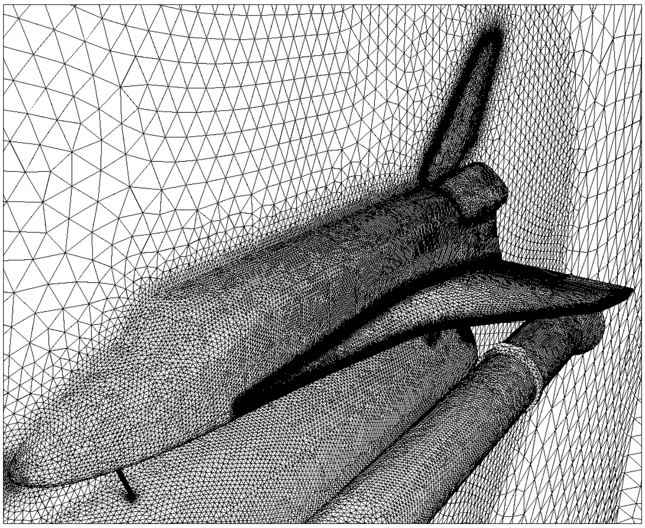
(a)



(b)

Figure 3.55. Space shuttle: (a) outline of domain; (b) close-up and sources; (c) surface mesh

slightly in the direction of the gravity vector. The objects that are at the ‘bottom’ will then be removed first from the active front and surrounded by new objects. The same technique can be applied if magnetic fields are present. Layered growth can be obtained be assigning a



(c)

Figure 3.55. Continued

slight increase in size based on the object number:

$$\delta = (1 + \epsilon n_{\mathrm{obj}})\delta_0, \tag{3.56}$$

where δ, δ_0, n_{obj} and ϵ denote the size used, original size, object number and a very small number (e.g. $\epsilon = 10^{-10}$), respectively. So-called radius growing, used to simulate granules or stone (Sakaguchi and Murakami (2002)), can be achieved by first generating a coarse cloud of objects, and then using layered growth around each of these.

Another interesting deposition pattern is the so-called bin or sieve placement, whereby spheres or objects of different average diameter are placed in space in such a way that the average number of spheres of a certain diameter in a given region of space corresponds to a given distribution. This deposition pattern can be implemented by placing the required distribution of sphere sizes into bins of decreasing sphere size. One then performs several passes over the domain, filling it with spheres or objects of a diameter and with a volume packing according to the bin specifications. As an example, for concrete one starts with spheres of the aggregate size. The next sphere size would be of diameter one-half to one-quarter of the original one, the next again one-half to one-quarter of the previous one, etc. Typically, no more than three to four bins are required. For each subsequent pass, all points/objects generated so far are considered part of the initial front. As the new spheres/objects have a much smaller size than all previous ones, each one of the spheres/objects generated in previous bin passes allows for sufficient ‘space’ for the generation of new spheres/objects.

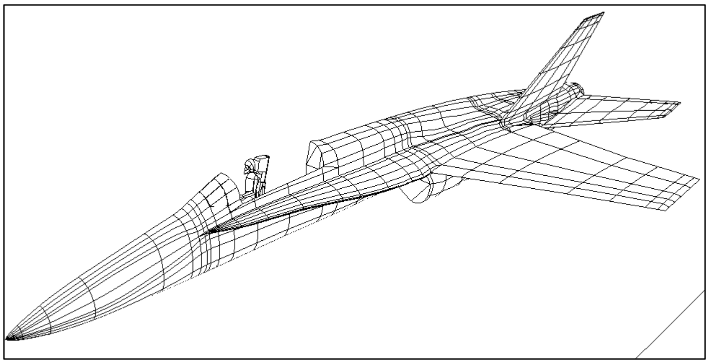
3.14. Applications

The following section shows several grids, as well as clouds of points and arbitrary objects, which were constructed using the techniques described in this chapter.

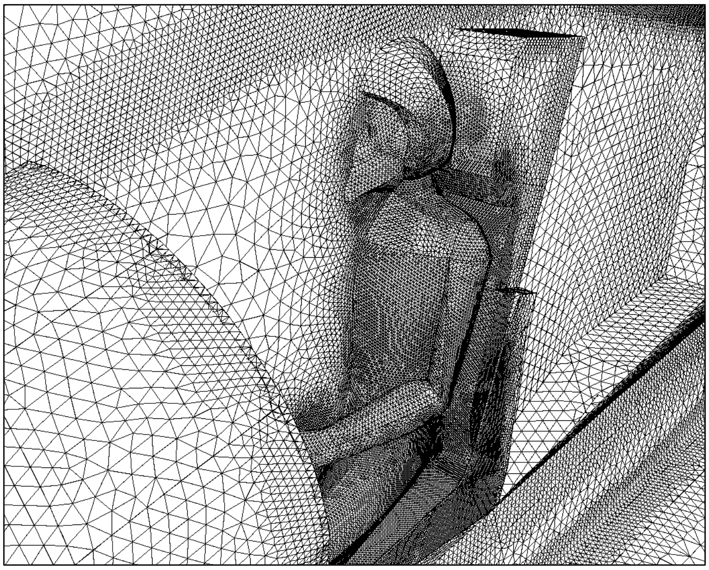
3.14.1. SPACE SHUTTLE ASCEND CONFIGURATION

This example is typical of many aerodynamic data sets. The outline of the domain, which was defined via 1271 surface patches, is shown in Figure 3.55(a). A close-up of the shuttle, together with some of the approximately 200 sources used to specify the desired element size in space, can be seen in Figure 3.55(b).

The surface triangulation of the final mesh, which had approximately 4 million tetrahedra, is shown in Figure 3.55(c). The smallest and largest specified element side lengths were 5.08 cm and 467.00 cm, respectively, i.e. an edge–length ratio of approximately $1{:}10^2$ and a volume ratio of $1{:}10^6$.



(a)



(b)

Figure 3.56. F18 pilot ejection: (a) outline of domain; (b) surface mesh (close-up)

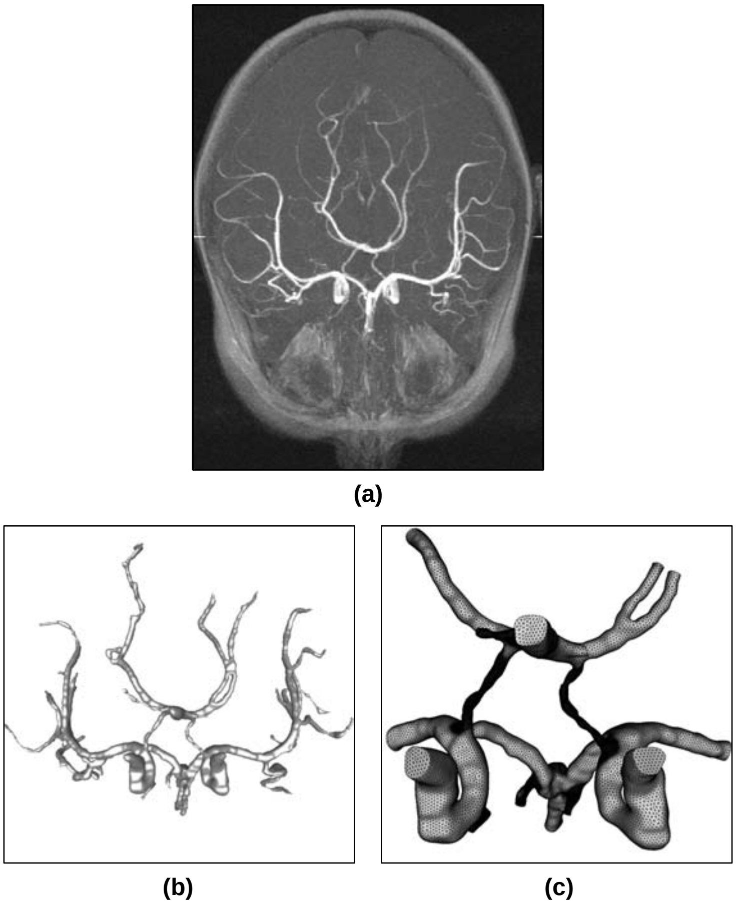
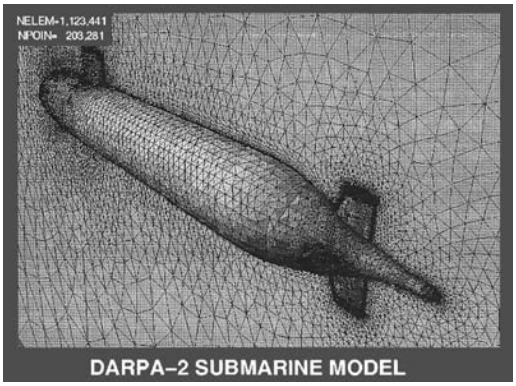


Figure 3.57. Circle of Willis: (a) medical images; (b) discrete surface definition; (c) surface mesh

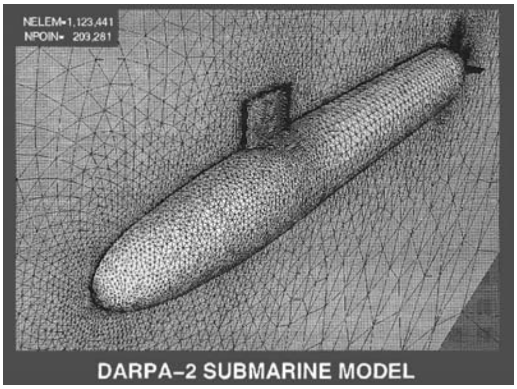
3.14.2. PILOT EJECTING FROM F18

Problems with moving bodies often require many re-meshings during the course of a simulation. The case shown here was taken from a pilot ejection simulation recently conducted (Sharov *et al.* (2000)). The outline of the domain, which consisted of 1345 patches, is shown in Figure 3.56(a).

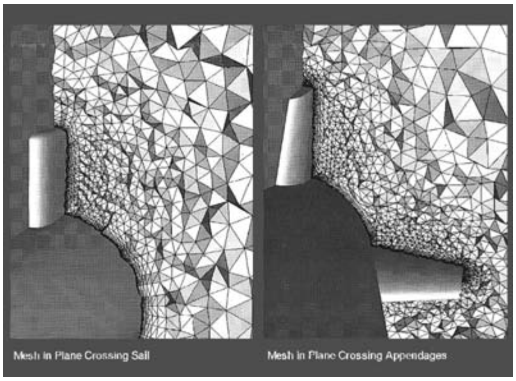
The surface triangulation of the final mesh, which had approximately 14 million tetrahedra, is shown in Figure 3.56(b). The smallest and largest specified element side lengths were 0.65 cm and 250.00 cm, respectively, i.e. an edge–length ratio of approximately $1{:}4 \times 10^2$ and a volume ratio of $1{:}5.6 \times 10^7$. The spatial variation of element size was specified via approximately 110 sources.



(a)



(b)


Mesh in Plane Crossing Sail
Mesh in Plane Crossing Appendages

(c)

Figure 3.58. DARPA-2 model: (a), (b) surface triangulation; (c) cross-sections in mesh

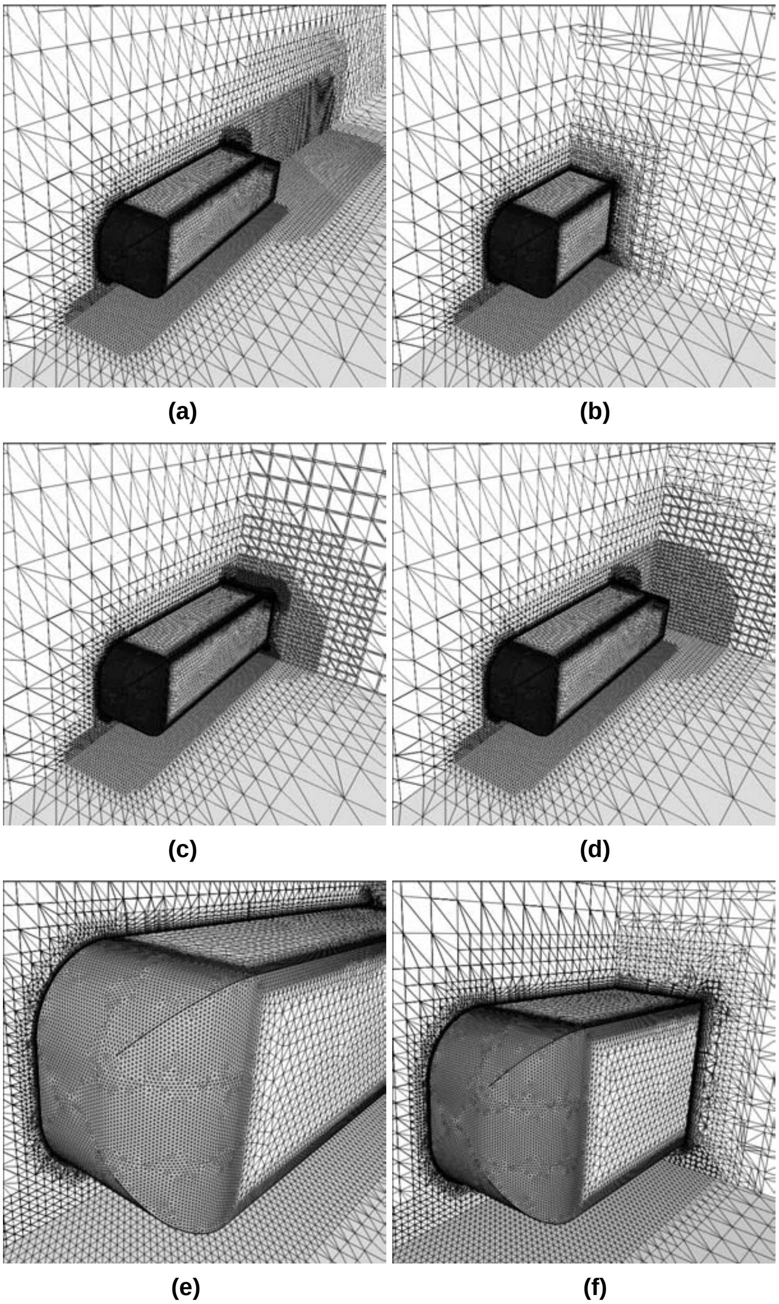


Figure 3.59. Ahmed Body: (a)–(d) mesh in different cuts; (e), (f) detail of mesh

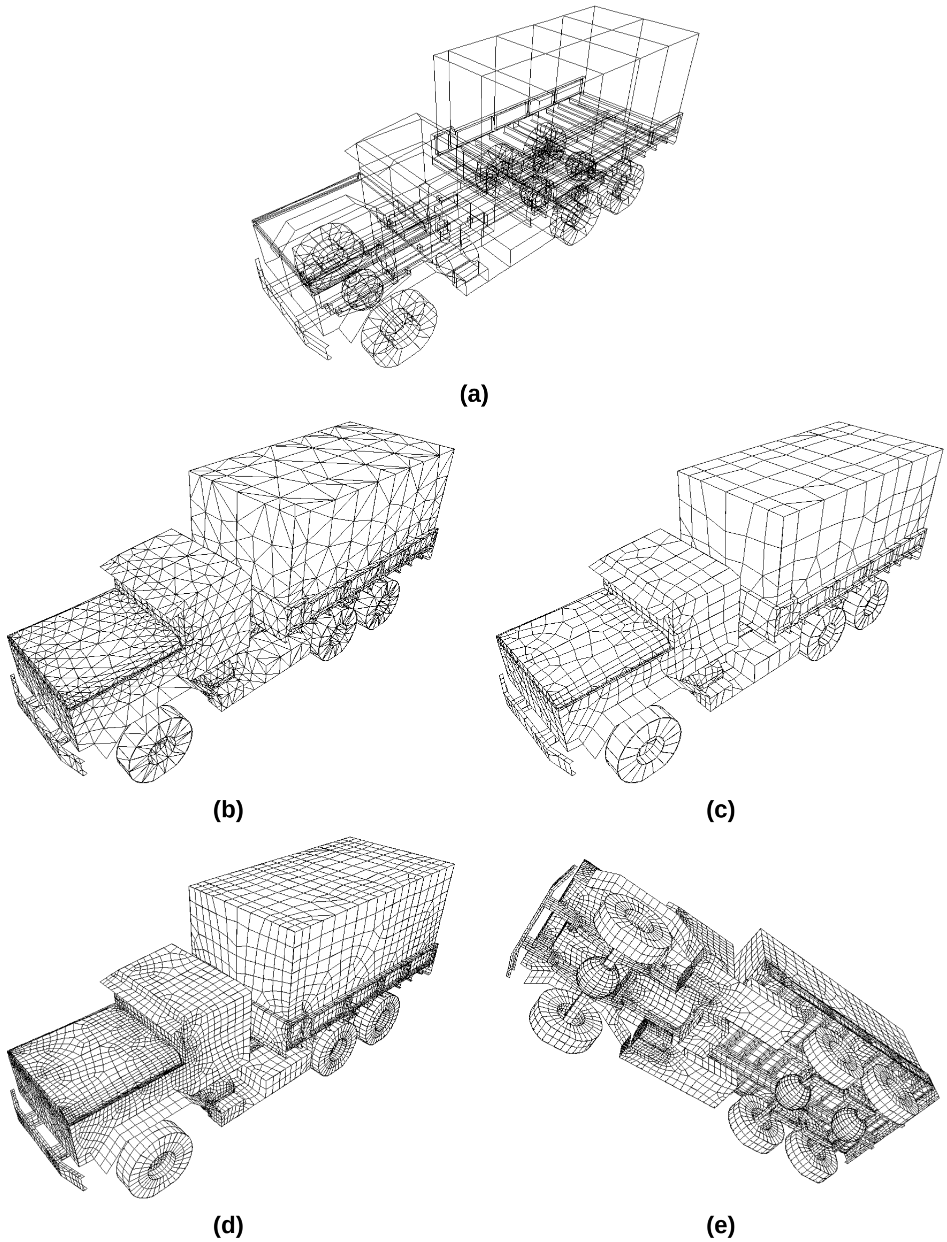


Figure 3.60. Truck: (a) CAD definition; (b) triangulation and (c) mixed quad/triangular mesh; (d), (e) final all-quad surface mesh

3.14.3. CIRCLE OF WILLIS

This example is typical of patient-specific biomedical CFD simulations (Cebral and Löhner (1999)). The surface of the domain is not specified via analytical patches, but via a

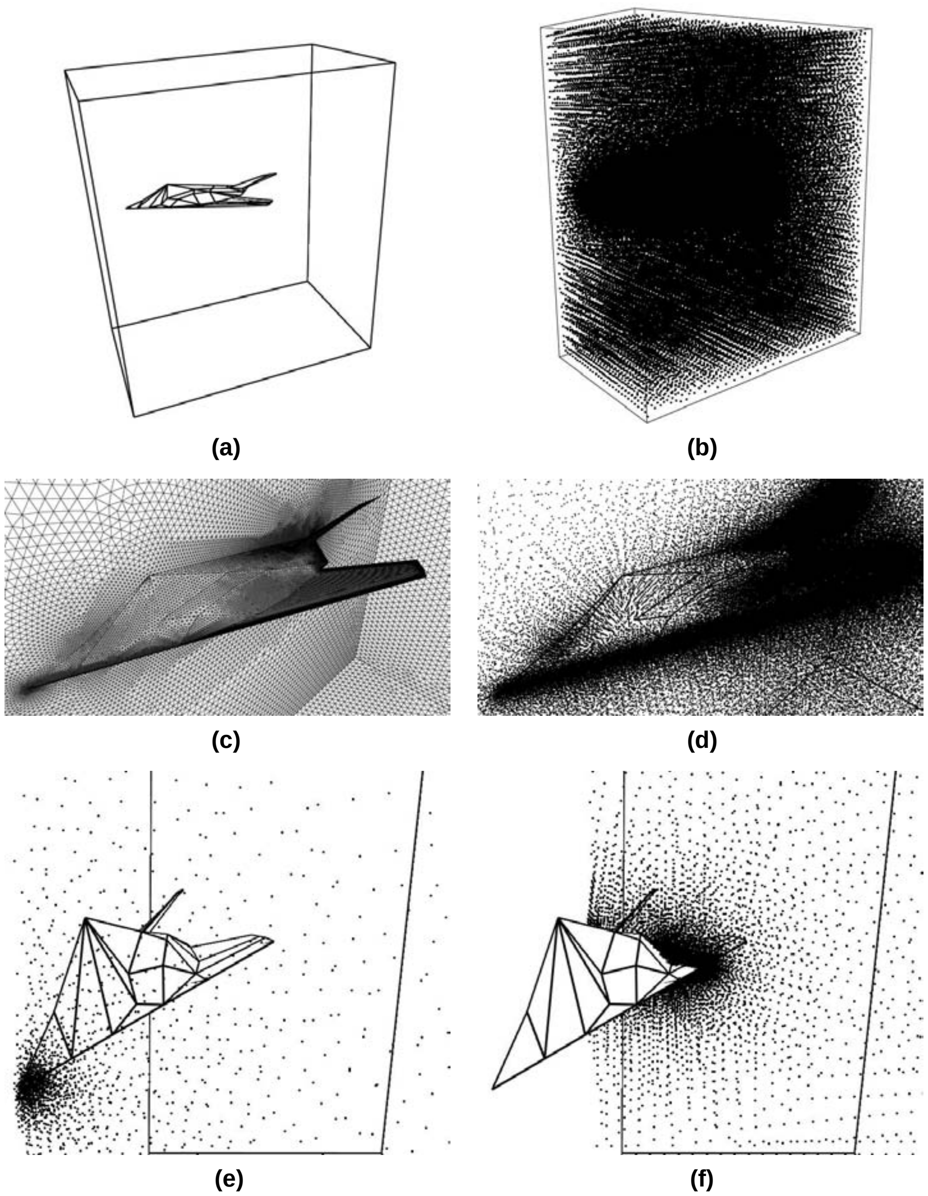


Figure 3.61. F117: (a) CAD definition; (b) global cloud of points; (c) close-up of surface mesh; (d) cloud of points; (e), (f) cuts at $x = 0, 120$; (g), (h) cuts at $x = 120$ (detail) and $x = 190$

triangulation. Figure 3.57(a) shows the maximum intensity projection of the medical images used as a starting point to generate the arterial surfaces. The (discrete) surface defining the outline of the CFD domain is shown in Figure 3.57(b). On this surface, the surface

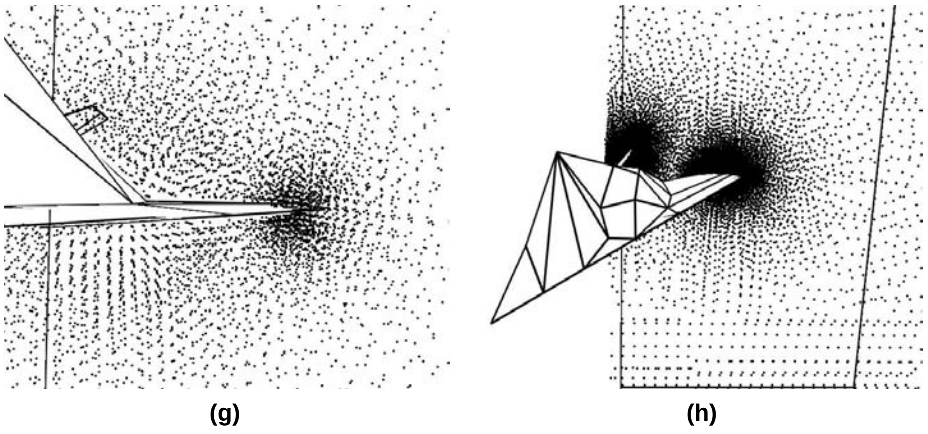


Figure 3.61. Continued

triangulation for a subsequent CFD run is generated (Figure 3.57(c)). The final mesh had approximately 3.1 million tetrahedra.

3.14.4. GENERIC SUBMARINE BODY

This example demonstrates the use of boundary-layer gridding techniques. A RANS grid for the generic DARPA-2 appended submarine model is considered. The triangulation of the surfaces on which RANS grids are required had 27 389 faces. The semi-structured grid generated from the surface triangulation consisted of 1 150 333 elements. Application of the removal criteria reduced this number to 682 401 elements. The fully unstructured grid generator then introduced another 441 040 elements. The surface of the final grid is given in Figures 3.58(a) and (b). A close-up of the semi-structured region is shown in Figure 3.58(c), where one can again see the smooth transition between semi-structured and unstructured grid regions.

3.14.5. AHMED CAR BODY

This example demonstrates the use of boundary layer gridding techniques, as well as tetrahedra obtained from Cartesian cores for the field. The geometry and discretization used are shown in Figure 3.59. One can clearly see the RANS grid, the adaptively refined Cartesian cores of the flow domain, as well as the transition to an unstructured grid between these two zones. The number of elements in the RANS region is `nelns=1,302K`, in the Cartesian core region `necrt=820K` and in the remaining unstructured grid region `nelun=415K`.

3.14.6. TRUCK

This example shows the generation of quad-shells on a truck configuration. The surface definition is shown in Figure 3.60(a). Based on this surface definition, a triangulation with elements of approximately twice the length as that desired for the final mesh is generated (Figure 3.60(b)). As many triangles as possible are merged into acceptable quads, and the

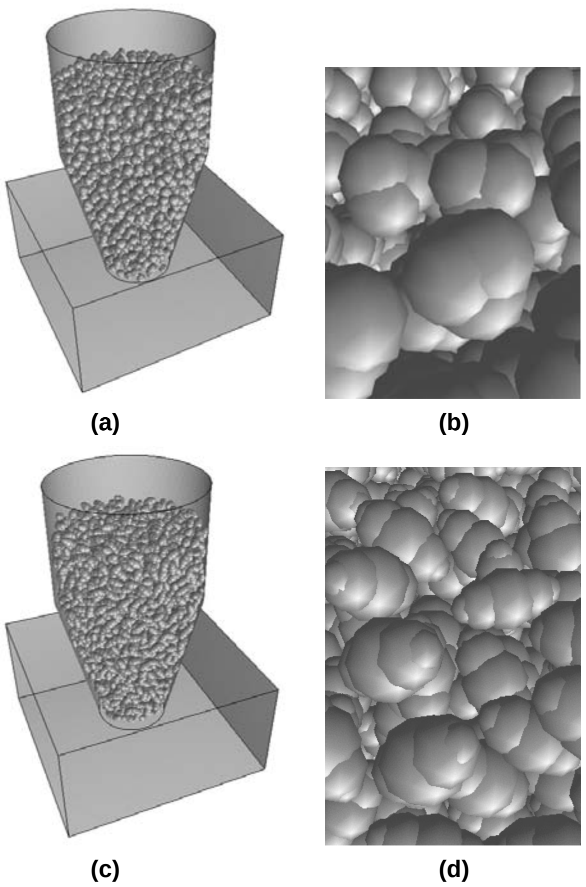


Figure 3.62. (a), (b) Hopper filled with beans; (c), (d) hopper filled with ellipsoids

mesh is smoothed (Figure 3.60(c)). One level of refinement yields the desired all-quad mesh, which is displayed in Figures 3.60(d) and (e). As one can see, the majority of quads are of very good quality.

3.14.7. POINT CLOUD FOR F117

A cloud of points for the aerodynamic simulation of inviscid, transonic flow past (half) an F117 fighter via FPM (Löhner *et al*. (2002)) is considered next. The point density was specified through the combination of a background grid and background sources. The surface triangulation consisted of approximately 50 000 points and 100 000 triangles. The advancing front point generator added another 200 000 points using simple stencil placement. Figure 3.61, taken from Löhner and Oñate (2004), shows the CAD definition of the computational domain, the global cloud of points, a close-up of the surface mesh, the cloud of points close to the plane, as well as some slices through the volume. The spatial variation of point density is clearly visible. The complete generation process (boundary triangulation, volume fill, output of files, etc.) took 44 s on a PC with Intel P4 chip running at 3.2 GHz, 1 Gbyte Ram, a Linux OS and an Intel Compiler.

3.14.8. HOPPER FILLED WITH BEANS/ELLIPSOIDS

Granular materials that require simulation include grains, ground stone, woodchips and many other materials (Cundall and Stack (1979), Cleary (1998)). Bridging in silos and hoppers can cause severe structural damage and has always been a concern. Figures 3.62(a) and (b) show a hopper configuration with bean-like objects composed of four spheres each. The total number of beans is 2124, i.e. 8496 spheres, and took 7 s to generate. Figures 3.62(c) and (d) show the same configuration filled with ellipsoidal objects composed of five spheres each. The total number of ellipsoids is 2794, i.e. 13 970 spheres, and took 10 s to generate.

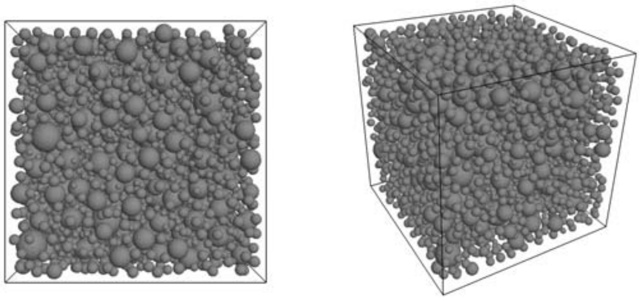


Figure 3.63. Cube filled with spheres of different sizes

3.14.9. CUBE FILLED WITH SPHERES OF DIFFERENT SIZES

Concrete and other aggregate materials often exhibit spherical or ellipsoidal objects of different sizes. The number of objects can be recorded in a statistical distribution of size versus the number of particles. Figure 3.63 shows a case for concrete: three different average sphere sizes, each on average half the size of the previous one and each with a standard (Gaussian) variation of 10%, are used to fill the cube shown. The total number of spheres is 3958.

4 APPROXIMATION THEORY

The ideas of approximation theory are at the root of any discretization scheme, be it finite difference, finite volume, finite element, spectral element or boundary element. Whenever a PDE is solved numerically, the exact solution is being approximated by some *known* set of functions. How this is accomplished is described in the next two chapters. The present chapter is devoted to the basic notions in approximation theory. Throughout the book the Einstein summation convention will be used. Every time an index appears twice in an equation, the summation over all possible values of the index is assumed.

4.1. The basic problem

Simply stated, the *basic problem* is the following.

Given a function $u(x)$ in the domain Ω, approximate $u(x)$ by *known* functions $N^i(x)$ and a set of free parameters a_i:

$$u(x) \approx u^h(x) = N^i(x) a_i. \tag{4.1}$$

The situation has been sketched in Figure 4.1.

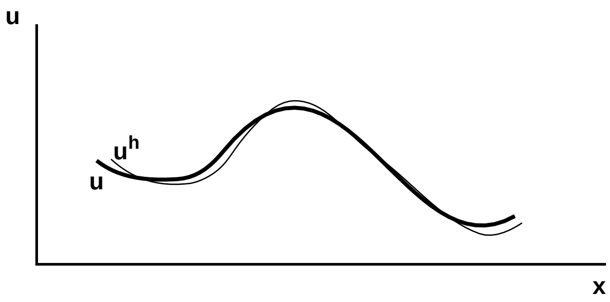


Figure 4.1. Approximation of functions

Examples of approximations that are often employed are:

- truncated Taylor series

$$u(x) \approx u^h(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_m x^m \Rightarrow a_j = \frac{1}{j!} \frac{d^j u}{dx^j}\bigg|_{x=0}; \tag{4.2a}$$

- Fourier expansions (e.g. truncated sine series)

$$u(x) \approx u^h(x) = a_j \sin \frac{j\pi x}{L} \Rightarrow a_j = \frac{2}{L} \int_0^L u(x) \sin \frac{j\pi x}{L}\, dx; \tag{4.2b}$$

Applied Computational Fluid Dynamics Techniques: An Introduction Based on Finite Element Methods, Second Edition.
Rainald Löhner © 2008 John Wiley & Sons, Ltd. ISBN: 978-0-470-51907-3

- Legendre polynomials;
- Hermite polynomials, etc.

In general, a ***complete*** set of trial functions N^j is chosen:

$$u(x) \approx u^h(x) = N^j a_j; \quad j = 1, 2, \ldots, m. \tag{4.3}$$

An open question that remains to be answered is how best to choose the unknown constants a_i in (4.1) and (4.3). Several possibilities are considered in what follows.

4.1.1. POINT FITTING

Here $u^h = u$ is enforced at m selected points in space:

$$u^h|_{x_k} = u|_{x_k}, \quad k = 1, 2, \ldots, m \Rightarrow N^i(x_k)a_j = u|_{x_k}. \tag{4.4}$$

This is by far the simplest choice, and has been sketched in Figure 4.2. The set of m conditions will lead to a linear system of coupled equations which may be represented in matrix notation as

$$\mathbf{K} \cdot \mathbf{a} = \mathbf{f}. \tag{4.5}$$

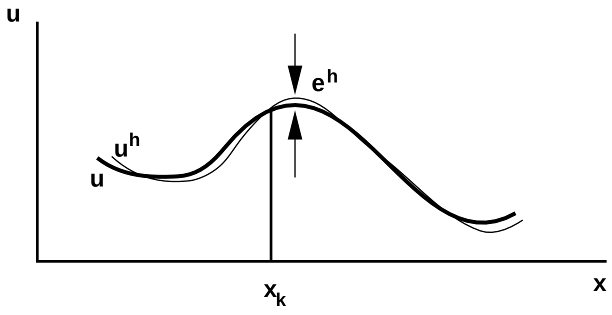


Figure 4.2. Point fitting

4.1.2. WEIGHTED RESIDUAL METHODS

Weighted residual methods (WRMs) are a more general class of methods, and we define the ***error*** $\epsilon^h = u - u^h$, and require

$$\epsilon^h(x) \to 0, \quad x \in \Omega. \tag{4.6}$$

Obviously ϵ^h does not vanish in most of the domain. In order for ϵ^h to be as small as possible, it must be penalized, or weighted, in some rational way. The easiest way to accomplish this is by introducing a set of ***weighting functions*** W^i, $i = 1, 2, \ldots, m$. The requirement given by (4.6) is restated as

$$\int_\Omega W^i \epsilon^h \, d\Omega = 0, \quad i = 1, 2, \ldots, m. \tag{4.7}$$

Then, given a complete set of trial and weighting functions, as $m \to \infty$ the error satisfies $\epsilon^h \to 0$ for all points in Ω. Inserting the expression for u^h yields

$$\int_\Omega W^i (u - N^j a_j)\, d\Omega = 0. \tag{4.8}$$

As before, the m integrals obtained for the different W^i-functions will lead to a linear system of coupled equations of the form

$$\mathbf{K} \cdot \mathbf{a} = \mathbf{r}, \tag{4.9a}$$

where

$$K^{ij} = \int_\Omega W^i N^j \, d\Omega, \quad r^i = \int_\Omega W^i u \, d\Omega. \tag{4.9b,c}$$

An important remark is that the choice of W^i defines the method. Two examples are included here to illustrate this.

(a) *Point collocation*: obtained by choosing

$$W^i = \delta(x - x_i), \quad x_i \in \Omega. \tag{4.10}$$

The WR statement becomes

$$\int_\Omega \delta(x - x_i)\epsilon^h \, d\Omega = \epsilon^h(x_i) = 0, \quad i = 1, 2, \ldots, m, \tag{4.11}$$

or

$$N^j(x_i)a_j = u(x_i). \tag{4.12}$$

Observe that this is the same as point fitting.

(b) *Galerkin method*: obtained by choosing

$$W^i = N^i. \tag{4.13}$$

The WR statement becomes

$$\int_\Omega N^i \epsilon^h \, d\Omega = \int_\Omega N^i (u - N^j a_j)\, d\Omega = 0, \quad i = 1, 2, \ldots, m, \tag{4.14a}$$

$$\left[\int_\Omega N^i N^j \, d\Omega\right] a_j = \int_\Omega N^i u \, d\Omega, \quad i = 1, 2, \ldots, m, \tag{4.14b}$$

or

$$\mathbf{M}_c \cdot \mathbf{a} = \mathbf{r}. \tag{4.14c}$$

The matrix $\mathbf{M}_c$ is the so-called consistent mass matrix. Observe that simply summing up the rows into the diagonal or ‘lumping’ of $\mathbf{M}_c$ does not lead to point fitting, as $\mathbf{r}$ contains the spatial variation in the integrals. The Galerkin choice of $W^i = N^i$ not only yields a symmetric matrix for $\mathbf{M}_c$, but also has some important properties which will be discussed in what follows.

4.1.3. LEAST-SQUARES FORMULATION

If we consider the least-squares problem

$$I_{ls} = \int_\Omega (\epsilon^h)^2 \, d\Omega = \int_\Omega (u^h - u)^2 \, d\Omega = \int_\Omega (N^k a_k - u)^2 \, d\Omega \to \min, \tag{4.15}$$

then it is known from the calculus of variations that the functional I_{ls} is minimized for

$$\delta I_{ls} = \delta a_k \int_\Omega N^k (N^l a_l - u) \, d\Omega = 0, \tag{4.16}$$

i.e. for each variable a_k

$$\int_\Omega N^k N^l \, d\Omega a_l = \int_\Omega N^k u \, d\Omega. \tag{4.17}$$

But this is the same as the Galerkin WRM! This implies that, of all possible choices for W^i, the Galerkin choice $W^i = N^i$ yields the best results for the least-squares norm I_{ls}. For other norms, other choices of W^i will be optimal. However, for the approximation problem, the norm given by I_{ls} seems the natural one (try to produce a counterexample).

4.2. Choice of trial functions

So far, we have dealt with general, global trial functions. Examples of this type of function family, or expansions, were the Fourier (sin-, cos-) and Legendre polynomials. For general geometries and applications, however, these functions suffer from the following drawbacks.

(a) Determining an appropriate set of trial functions is difficult for all but the simplest geometries in two and three dimensions.

(b) The resulting matrix **K** is full.

(c) The matrix **K** can become ill-conditioned, even for simple problems. A way around this problem is the use of strongly orthogonal polynomials. As a matter of fact, most of the global expansions used in engineering practice (Fourier, Legendre, etc.) are strongly orthogonal.

(d) The resulting coefficients a_j have no physical significance.

The way to circumvent all of these difficulties is to go from *global* trial functions to *local* trial functions. The domain Ω on which $u(x)$ is to be approximated is subdivided into a set of non-overlapping sub-domains Ω_{el} called *elements*. The approximation function u^h is then defined in each sub-domain separately. The situation is shown in Figure 4.3.

In what follows, we consider several possible choices for local trial functions.

4.2.1. CONSTANT TRIAL FUNCTIONS IN ONE DIMENSION

Consider the piecewise constant function, shown in Figure 4.4:

$$P^E = \begin{cases} 1 & \text{in element } E, \\ 0 & \text{in all other elements.} \end{cases} \tag{4.18}$$

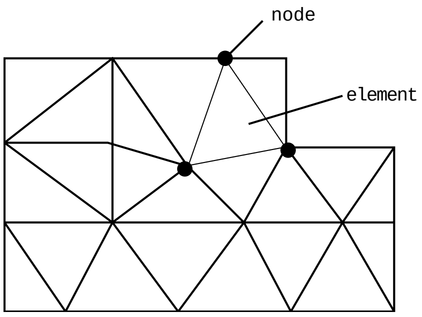


Figure 4.3. Subdivision of a domain into elements

Then, globally, we have

$$u \approx u^h = P^E u_E, \tag{4.19}$$

and locally, on each element *el*,

$$u \approx u^h = u_{el}. \tag{4.20}$$

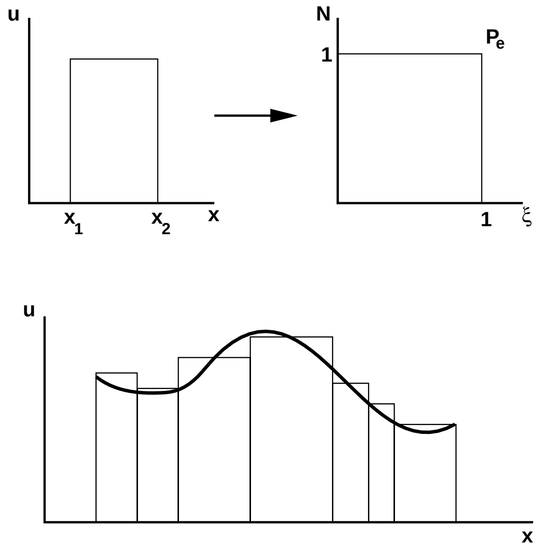


Figure 4.4. Piecewise constant trial function in one dimension

4.2.2. LINEAR TRIAL FUNCTIONS IN ONE DIMENSION

A better approximation is obtained by letting u^h vary linearly in each element. This is accomplished by placing nodes at the beginning and end of each element, and defining a piecewise linear trial function:

$$N^j = \begin{cases} 1 & \text{at node } j, \\ 0 & \text{at all other nodes,} \end{cases} \tag{4.21}$$

and N^j is non-zero only on elements associated with node j. Then, globally, we have

$$u \approx u^h = N^j(x)u(x_j) = N^j(x)\hat{u}_j, \tag{4.22}$$

and locally over element *el* with nodes 1 and 2

$$u \approx u^h = N^1\hat{u}_1 + N^2\hat{u}_2, \tag{4.23}$$

where

$$N^1_{el} = \frac{x_2 - x}{x_2 - x_1} = 1 - \xi, \quad N^2_{el} = \frac{x - x_1}{x_2 - x_1} = \xi. \tag{4.24}$$

Such a linear trial function is shown in Figure 4.5. Observe that

$$x = (1 - \xi)x_1 + \xi x_2 = N^1 x_1 + N^2 x_2, \tag{4.25}$$

implying that, if we consider the spatial extent of an element, it may also be mapped using the same trial functions and the coordinates of the endpoints.

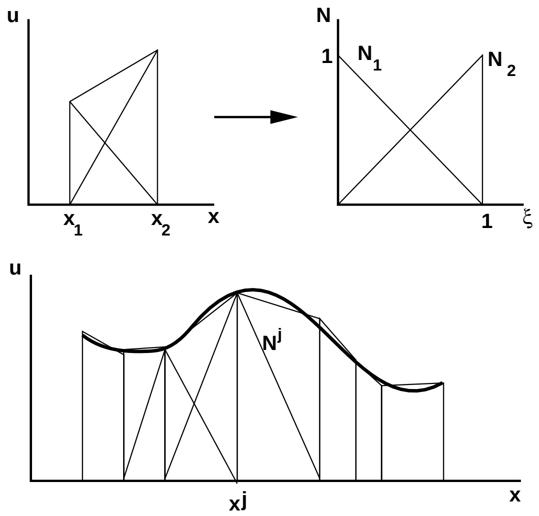


Figure 4.5. Piecewise linear trial function in one dimension

4.2.3. QUADRATIC TRIAL FUNCTIONS IN ONE DIMENSION

An even better approximation is obtained by letting u^h vary quadratically in each element. This may be achieved by placing nodes at the beginning and end of each element, as well as the middle, as shown in Figure 4.6.

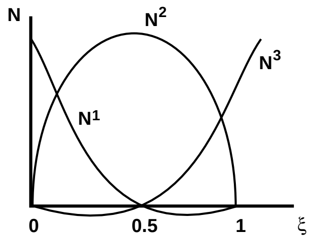


Figure 4.6. Piecewise constant trial function in one dimension

The resulting shape functions are of the form

$$N^1 = (1-\xi)(1-2\xi), \quad N^2 = 4\xi(1-\xi), \quad N^3 = -\xi(1-2\xi), \tag{4.26}$$

where ξ is given by (4.24).

4.2.4. LINEAR TRIAL FUNCTIONS IN TWO DIMENSIONS

A general linear function in two dimensions is given by the form

$$f(x,y) = a + bx + cy. \tag{4.27}$$

We therefore have three unknowns (a, b, c), requiring three nodes for a general representation. The natural geometric object with three nodes is the triangle.

The shape functions may be derived by observing from Figure 4.7 the map from Cartesian to local coordinates:

$$\mathbf{x} = \mathbf{x}_A + (\mathbf{x}_B - \mathbf{x}_A)\xi + (\mathbf{x}_C - \mathbf{x}_A)\eta, \tag{4.28}$$

or

$$\mathbf{x} = N^i \mathbf{x}_i = (1-\xi-\eta)\mathbf{x}_A + \xi\mathbf{x}_B + \eta\mathbf{x}_C, \tag{4.29}$$

implying, with the area coordinates $\zeta_1, \zeta_2, \zeta_3$ shown in Figure 4.8,

$$N^1 = \zeta_1 = 1-\xi-\eta, \quad N^2 = \zeta_2 = \xi, \quad N^3 = \zeta_3 = \eta. \tag{4.30}$$

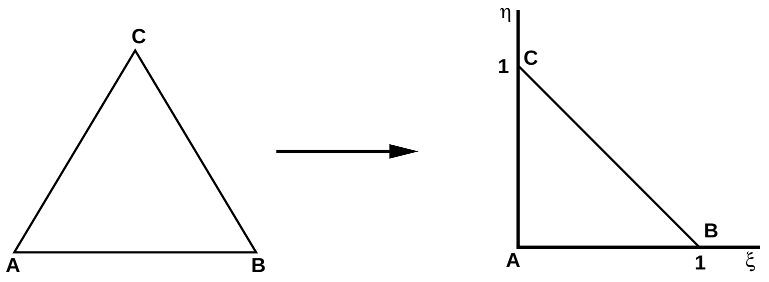


Figure 4.7. Linear triangle

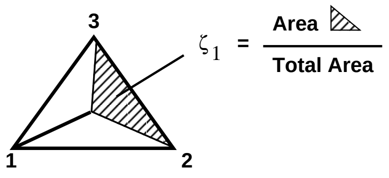


Figure 4.8. Area coordinates

The shape function derivatives, which are constant over the element, can be derived analytically by making use of the chain rule and the derivatives with respect to the local coordinates,

$$N^i_{,x} = N^i_{,\xi}\xi_{,x} + N^i_{,\eta}\eta_{,x}. \tag{4.31}$$

From (4.29), the Jacobian matrix of the derivatives is given by

$$\mathbf{J} = \begin{pmatrix} x,_\xi & x,_\eta \\ y,_\xi & y,_\eta \end{pmatrix} = \begin{pmatrix} x_{BA} & x_{CA} \\ y_{BA} & y_{CA} \end{pmatrix}, \tag{4.32}$$

its determinant by

$$\det(\mathbf{J}) = 2A_{el} = x_{BA}y_{CA} - x_{CA}y_{BA} \tag{4.33}$$

and the inverse

$$\mathbf{J}^{-1} = \begin{pmatrix} \xi,_x & \xi,_y \\ \eta,_x & \eta,_y \end{pmatrix} = \frac{1}{2A}\begin{pmatrix} y_{CA} & -x_{CA} \\ -y_{BA} & x_{BA} \end{pmatrix}. \tag{4.34}$$

We are now in a position to express the derivatives of the shape functions with respect to x, y analytically,

$$\begin{bmatrix} N^1 \\ N^2 \\ N^3 \end{bmatrix}_{,x} = \frac{1}{2A}\begin{bmatrix} -y_{CA} + y_{BA} \\ y_{CA} \\ -y_{BA} \end{bmatrix}, \tag{4.35a}$$

$$\begin{bmatrix} N^1 \\ N^2 \\ N^3 \end{bmatrix}_{,y} = \frac{1}{2A}\begin{bmatrix} x_{CA} - x_{BA} \\ -x_{CA} \\ x_{BA} \end{bmatrix}. \tag{4.35b}$$

Before going on, we derive some useful geometrical parameters that are often used in CFD codes.

From Figure 4.9, we can see that the *direction* of each normal is given by

$$\mathbf{n}^i = -\frac{\nabla N^i}{|\nabla N^i|}, \tag{4.36}$$

the *height* of each normal by

$$|\nabla N^i| = \frac{1}{h_i} \Rightarrow h_i = \frac{1}{|\nabla N^i|} \tag{4.37}$$

and the *face-normal* (area of face in direction of normal) by

$$(s\mathbf{n})^i = -s^i\frac{\nabla N^i}{|\nabla N^i|} = -s^i h_i \nabla N^i = -2A\nabla N^i \tag{4.38}$$

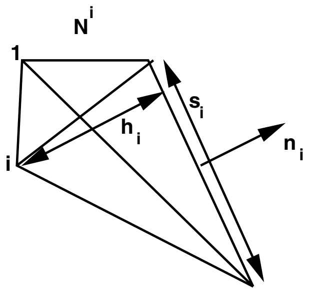


Figure 4.9. Geometrical properties of linear triangles

(no summation over i). The basic integrals amount to

$$\int_{el} N^i \, d\Omega = \frac{A}{3} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, \tag{4.39}$$

and

$$\mathbf{M}_{el} = \int N^i N^j \, d\Omega = \frac{\Delta_{el}}{12} \begin{bmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{bmatrix}. \tag{4.40}$$

4.2.5. QUADRATIC TRIAL FUNCTIONS IN TWO DIMENSIONS

A general quadratic function in two dimensions is given by the form

$$f(x, y) = a + bx + cy + dx^2 + exy + fy^2. \tag{4.41}$$

We therefore have six unknowns (a, b, c, d, e, f), requiring six nodes for a general representation. One possibility to represent these degrees of freedom is the six-noded triangle. Three nodes are positioned at the vertices and three along the faces (see Figure 4.10).

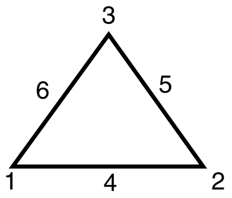


Figure 4.10. Quadratic triangle

The shape functions for this type of element are given by

$$\begin{aligned}
N^1 &= \zeta_1(2\zeta_1 - 1) = (1 - \xi - \eta)(1 - 2\xi - 2\eta),\\
N^2 &= \zeta_2(2\zeta_2 - 1) = \xi(2\xi - 1),\\
N^3 &= \zeta_3(2\zeta_3 - 1) = \eta(2\eta - 1),\\
N^4 &= 4\zeta_1\zeta_2 = 4\xi(1 - \xi - \eta),\\
N^5 &= 4\zeta_2\zeta_3 = 4\xi\eta,\\
N^6 &= 4\zeta_1\zeta_3 = 4\eta(1 - \xi - \eta).
\end{aligned} \tag{4.42}$$

One can see the correspondence between the 1-D shape functions for a quadratic element and the functions $N^1 - N^6$.

4.3. General properties of shape functions

All commonly used shape functions satisfy the following general properties:

(a) *Interpolation property*. Given that $u^h = N^i(x)\hat{u}_i$, we must have

$$u^h(x_j) = N^i(x_j)\hat{u}_i = \hat{u}_j \Rightarrow N^i(x_j) = \delta^i_j. \tag{4.43}$$

In more general terms, this is just a re-expression of the definition of local trial functions.

(b) *Constant sum property*. At the very least, any given set of trial functions must be able to represent a constant, e.g. $c = 1$. Therefore

$$u = 1 \Rightarrow u^h = 1 = N^i(x)\hat{u}_i. \tag{4.44}$$

But the interpolation property implies $\hat{u}_i = 1$, or

$$\sum_i N^i(x) = 1, \quad \forall x \in \Omega. \tag{4.45}$$

Thus, the sum of all shape functions at any given location equals unity.

(c) *Conservation property*. Given the constant sum property, it is easy to infer that

$$\sum_i N^i_{,k} = 0, \quad \forall x \in \Omega_{el}. \tag{4.46}$$

Thus, the sum of derivatives of all shape functions at any given location in the element vanishes.

4.4. Weighted residual methods with local functions

After this small excursion to define commonly used local trial functions and some of their properties, we return to the basic approximation problem. The WRM can be restated with local trial functions as follows:

$$\int_\Omega W^i(u - N^j\hat{u}_j)\, d\Omega = 0. \tag{4.47}$$

The *basic idea* is to split any integral that appears into a sum over the sub-domains or elements:

$$\int_{\Omega} \cdots d\Omega = \sum_{el} \int_{\Omega_{el}} \cdots d\Omega_{el}. \tag{4.48}$$

The integrals are then evaluated on a sub-domain or element level. At the element level, the definition of the trial functions is easily accomplished. The only information required is that of the nodes belonging to each element. The basic paradigm, which carries over to the solution of PDEs later on, is the following:

- gather information from global point arrays to local element arrays;
- build integrals on the element level;
- scatter-add resulting integrands to global right-hand side (rhs)/matrix locations.

This may be expressed mathematically as follows:

$$K^{ij} u_j = \left[\sum_{el} K^{ij}_{el}\right][u_j]_{el} = \sum_{el} r^i_{el} = r^i. \tag{4.49}$$

It is this basic paradigm that makes simple finite element or finite volume codes based on unstructured grids possible. As all the information that is required is that of the nodes belonging to an element, a drastic simplification of data structures and logic is achieved. Granted, the appearance of so many integrals can frighten away many an engineer. However, compared to the Bessel, Hankel and Riemann expansions used routinely by engineers only a quarter of a century ago, they are very simple.

4.5. Accuracy and effort

Consider the interesting question: What is the *minimum* order of approximation required for the trial functions in order to achieve vanishing errors without an infinite amount of work? In order to find an answer, let us assume that the present mesh already allows for a uniform (optimal) distribution of the error. Let us suppose further that we have a way to solve for the unknown coefficients $\hat{u}$ in linear time complexity, i.e. it takes $O(N)$ time to solve for the N unknowns (certainly a lower bound). Then the effort E_{ff} will be given by

$$E_{ff} \geq c_1 h^{-d}, \tag{4.50}$$

where d is the dimensionality of the problem and h a characteristic element length. On the other hand, the error is given by

$$\epsilon^h = \|u - u^h\| = c_2 h^{p+1} |u|_{p+1}, \tag{4.51}$$

where p is the order of approximation for the elements. We desire to attain $\|u - u^h\| \to 0$ *faster than* $E_{ff} \to \infty$. Thus, we desire

$$\lim_{h\to 0} E_{ff} \cdot \|u - u^h\| = \lim_{h\to 0} c_3 h^{p+1-d} |u|_{p+1} \to 0. \tag{4.52}$$

Table 4.1. Accuracy and effort

Dimension	$E_{ff} \cdot \epsilon^h$	Decrease with $h \to 0$
1-D	h^p	$p \geq 1$
2-D	h^{p-1}	$p \geq 2$
3-D	h^{p-2}	$p \geq 3$

Consider now the worst-case scenario (e.g. homogeneous turbulence). The resulting order of approximation for break even is given in Table 4.1.

Table 4.1 indicates that one should strive for schemes of higher order as the dimensionality of the problem increases. Given that most current CFD codes are of second-order accuracy, we have to ask why they are still being used for 3-D problems.

(a) The first and immediate answer is, of course, that most code writers simply transplanted their 1-D schemes to two and three dimensions without further thought.

(b) A second answer is that the analysis performed above only holds for very high-accuracy calculations. In practice, we do not know turbulence viscosities to better than 10% locally, and in most cases even laminar viscosity or other material properties to better than 1%, so it is not necessary to be more accurate.

(c) A third possible answer is that in many flow problems the required resolution is not the same in all dimensions. Most flows have lower-dimensional features, like boundary layers, shocks or contact discontinuities, embedded in 3-D space. It is for this reason that second-order schemes still perform reasonably well for engineering simulations.

(d) A fourth answer is that high-order approximation functions also carry an intrinsic, and very often overlooked, additional cost. For a classic finite difference stencil on a Cartesian grid the number of neighbour points required increases linearly with the order of the approximation (three-point stencils for second order, five-point stencils for fourth order, seven-point stencils for sixth order, etc.), i.e. the number of off-diagonal matrix coefficients will increase as $w_{\text{dof}}^{FD} = pd$ (as before, d denotes the dimensionality of the problem and p the order of the approximation). On the other hand, for general high-order approximation functions all entries of the mass matrix $\mathbf{K}$ in (4.9) have to be considered at the element level. Consider for the sake of simplicity Lagrange polynomials of order p in tensor-product form for 1-D, 2-D (quadrilateral) and 3-D (hexahedral) elements. The number of degrees of freedom in these elements will increase according to $n_{\text{dof}} = (1 + p)^d$, implying $n_k = (1 + p)^{2d}$ matrix entries. The matrix–vector product, which is at the core of any efficient linear equation solver (e.g. multigrid) therefore requires $O(1 + p)^{2d}$ floating point operations, i.e. the work per degree of freedom is of $w_{\text{dof}} = O(1 + p)^d$. The resulting cost per degree of freedom, as well as the relative cost C_r as compared to linear elements, is summarized in Table 4.2 for 3-D Lagrange elements and Finite Differences (FD3D). Note the marked increase in cost with the order of approximation. These estimates assume that the system matrix $\mathbf{K}$ only needs to be built once, something that will not be the case for nonlinear operators (one can circumvent this restriction if one approximates the fluxes as well, see Atkins and Shu (1996)). As the matrix entries can no longer be evaluated analytically, an additional cost factor proportional to the number of Gauss points $n_g = O(1+p)^d$ will be incurred. The work per degree of freedom thus becomes $w^{nl}_{\rm dof} = O(1+p)^{2d}$.

Table 4.2. Effort per degree of freedom as a function of the approximation

$w^{3\text{-}D}_{\rm dof}$	$C^{3\text{-}D}_r$	$w^{nl3\text{-}D}_{\rm dof}$	$C^{nl3\text{-}D}_r$	$w^{FD3\text{-}D}_{\rm dof}$	$C^{FD3\text{-}D}_r$	
1	8	1.0	64	1.0	6	1.0
2	27	3.4	729	11.4	12	2.0
3	64	8.0	4096	64.0	18	3.0
4	125	15.6	15 625	244.1	24	4.0
5	216	27.0	46 656	729.0	30	5.0
6	343	42.9	117 649	1839.3	36	6.0

If we focus in particular on the comparison of work for nonlinear operators, we see that the complete refinement of a mesh of linear elements, which increases the number of elements by a factor of 1:2^d does not appear as hopelessly uncompetitive as initially assumed.

4.6. Grid estimates

Let us consider the meshing (and solver) requirements for typical aerodynamic and hydrodynamic problems purely from the approximation theory standpoint. Defining the Reynolds number as

$$Re = \frac{\rho |\mathbf{v}_\infty| l}{\mu}, \tag{4.53}$$

where ρ, $\mathbf{v}$, μ and l denote the density, free stream velocity and viscosity of the fluid, as well as a characteristic object length, respectively, we have the following estimates for the boundary-layer thickness and gradient at the wall for flat plates from boundary-layer theory (Schlichting (1979)):

(a) *Laminar flow*:

$$\frac{\delta(x)}{x} = 5.5 Re_x^{-1/2}, \qquad \left.\frac{\partial v}{\partial y}\right|_{y=0} = 0.332 Re_x^{1/2}; \tag{4.54}$$

(b) *Turbulent flow*:

$$\frac{\delta(x)}{x} = 5.5 Re_x^{-1/5}, \qquad \left.\frac{\partial v}{\partial y}\right|_{y=0} = 0.0288 Re_x^{4/5}. \tag{4.55}$$

This implies that the minimum element size required to capture the main vortices of the boundary layer (via a large-eddy simulation (LES)) will be $h \approx Re^{-1/2}$ and $h \approx Re^{-1/5}$ for the laminar and turbulent cases. In order to capture the laminar sublayer (i.e. the wall gradient, and hence the friction) properly, the first point off the wall must have a (resolved) velocity that is only a fraction of the free-stream velocity:

$$\left.\frac{\partial v}{\partial y}\right|_{y=0} h_w = \epsilon v_\infty, \tag{4.56}$$

Table 4.3. Estimate of grid and timestep requirements

Simulation type	`npoin`	`ntime`
Laminar	$10^4\ Re$	$10^2\ Re^{1/2}$
VLES	$10^4\ Re^{2/5}$	$10^2\ Re^{1/5}$
LES	$10^6\ Re^{2/5}$	$10^3\ Re^{1/5}$
DNS	$10^4\ Re^{8/5}$	$10^2\ Re^{4/5}$

Table 4.4. Estimate of grid and timestep requirements

Re	n_p VLES	n_t VLES	n_p LES	n_t LES	n_p DNS	n_t DNS
10^6	$10^{6.4}$	$10^{3.2}$	$10^{8.4}$	$10^{4.2}$	$10^{13.6}$	$10^{6.8}$
10^7	$10^{6.8}$	$10^{3.4}$	$10^{8.8}$	$10^{4.4}$	$10^{15.2}$	$10^{7.6}$
10^8	$10^{7.2}$	$10^{3.6}$	$10^{9.2}$	$10^{4.6}$	$10^{16.8}$	$10^{8.4}$
10^9	$10^{7.6}$	$10^{3.8}$	$10^{9.6}$	$10^{4.8}$	$10^{18.4}$	$10^{9.2}$

implying that the element size close to the wall must be inversely proportional to the gradient. This leads to element size requirements of $h \approx Re^{-1/2}$ and $h \approx Re^{-4/5}$, i.e. considerably higher for the turbulent case. Let us consider in some more detail a wing of aspect ratio Λ. As is usual in aerodynamics, we will base the Reynolds number on the root chord length of the wing. We suppose that the element size required near the wall of the wing will be of the form (see above)

$$h \approx \frac{1}{\alpha Re^q}, \tag{4.57}$$

and that, at a minimum, β layers of this element size will be required in the direction normal to the wall. If we assume, conservatively, that most of the points will be in the (adaptively/optimally gridded) near-wall region, the total number of points will be given by

$$n_p = \Lambda\beta\alpha^2 Re^2 q. \tag{4.58}$$

Assuming we desire an accurate description of the vortices in the flowfield, the (significant) advective time scales will have to be resolved with an explicit time-marching scheme. The number of timesteps required will then be at least proportional to the number of points in the chord direction, i.e. of the form

$$n_t = \frac{\gamma}{h} = \alpha\gamma Re^q. \tag{4.59}$$

Consider now the best-case scenario: $\alpha = \beta = \gamma = \Lambda = 10$. In the following, we will label this case the ‘Very Large Eddy Simulation’ (VLES). A more realistic set of numbers for typical LES simulations would be: $\alpha = 100$, $\beta = \gamma = \Lambda = 10$. The number of points required for simulations based on these estimates are summarized in Tables 4.3 and 4.4. Recall that the Reynolds number for cars and trucks lies in the range $Re = 10^6$–10^7, for aeroplanes $Re = 10^7$–10^8, and for naval vessels $Re = 10^8$–10^9. At present, any direct simulation of Navier–Stokes (DNS) is out of the question for the Reynolds numbers encountered in aerodynamic and hydrodynamic engineering applications.

5 APPROXIMATION OF OPERATORS

While approximation theory dealt with the problem

$$\text{Given } u, \text{ approximate } \|u - u^h\| \rightarrow \min,$$

the numerical solution of differential or integral equations deals with the following problem:

$$\text{Given } L(u) = 0, \text{ approximate } \|L(u) - L(u^h)\| \rightarrow \min \Rightarrow \|L(u^h)\| \rightarrow \min.$$

Here $L(u)$ denotes an operator, e.g. the Laplace operator $L(u) = \nabla^2 u$. The aim is to minimize the error of the operator using known functions

$$\epsilon_L^h = L(u^h) = L(N^i \hat{u}_i) \rightarrow 0. \tag{5.1}$$

As before, the method of weighted residuals represents the most general way in which this minimization may be accomplished

$$\int_\Omega W^i \epsilon_L^h \, d\Omega = 0, \quad i = 1, 2, \ldots, m. \tag{5.2}$$

5.1. Taxonomy of methods

The choice of trial and test functions N^i, W^i defines the method. Since a large amount of work has been devoted to some of the more successful combinations of N^i, W^i, a classification is necessary.

5.1.1. FINITE DIFFERENCE METHODS

Finite difference methods (FDMs) are obtained by taking N^i polynomial, $W^i = \delta(x_i)$. This implies that for such methods $\epsilon_L^h = L(u^h) = 0$ is enforced at a finite number of locations in space. The choices of polynomials for N^i determine the accuracy or order of the resulting discrete approximation to $L(u)$ (Collatz (1966)). This discrete approximation is referred to as a *stencil*. FDMs are commonly used in CFD for problems that exhibit a moderate degree of geometrical complexity, or within multiblock solvers. The resulting stencils are most easily derived for structured grids with uniform element size h. For this reason, in most codes based on FDMs, the physical domain, as well as the PDE to be solved (i.e. $L(u)$), are transformed to a square (2-D) or cube (3-D) that is subsequently discretized by uniform elements.

5.1.2. FINITE VOLUME METHODS

Finite volume methods (FVMs) are obtained by taking N^i polynomial, $W^i = 1$ if x$\subset \Omega_{el}$, 0 otherwise. As W^i is constant in each of the respective elements, any integrations by part reduce to element boundary integrals. For first-order operators of the form

$$L(u) = \nabla \cdot \mathbf{F}(u), \tag{5.3}$$

this results in

$$\int_{\Omega} W^i L(u)\, d\Omega = \int_{\Omega_{el}} \nabla \cdot \mathbf{F}(u)\, d\Omega_{el} = -\int_{\Gamma_{el}} \mathbf{n} \cdot \mathbf{F}(u)\, d\Gamma_{el}. \tag{5.4}$$

This implies that only the normal fluxes through the element faces $\mathbf{n} \cdot \mathbf{F}(u)$ appear in the discretization. FVMs are commonly used in CFD in conjunction with structured and unstructured grids. For operators with second-order derivatives, the integration is no longer obvious, and a number of strategies have been devised to circumvent this limitation. One of the more popular ones is to evaluate the first derivatives in a first pass over the mesh, and to obtain the second derivatives in a subsequent pass.

5.1.3. GALERKIN FINITE ELEMENT METHODS

In Galerkin FEMs (GFEMs), N^i is chosen as a polynomial, and $W^i = N^i$. This special choice is best suited for operators that may be derived from a minimization principle. It is widely used for thermal problems, structural dynamics, potential flows and electrostatics (Zienkiewicz and Morgan (1983), Zienkiewicz and Taylor (1988)).

5.1.4. PETROV–GALERKIN FINITE ELEMENT METHODS

Petrov–Galerkin FEMs (PGFEMs) represent a generalization of GFEMs. Both N^i, W^i are taken as polynomials, but $W^i \neq N^i$. For operators that exhibit first-order derivatives, PGFEMs may be superior to GFEMs. On the other hand, once GFEMs are enhanced by adding artificial viscosities and background damping, the superiority is lost.

5.1.5. SPECTRAL ELEMENT METHODS

Spectral element methods (SEMs) represent a special class of FEMs. They are distinguished from all previous ones in that they employ, locally, special polynomials or trigonometric functions for N^i in order to avoid the badly conditioned matrices that would arise for higher-order Lagrange polynomials. The weighting functions can be either $W^i = \delta(x_i)$ (so-called collocation), or $W^i = N^i$. Special integration or collocation rules further set this class of methods apart from GFEMs.

5.2. The Poisson operator

Let us now exemplify the use of the GFEM on a simple operator. The operator chosen is the Poisson operator. Given

$$\nabla^2 u - f = 0 \quad \text{in } \Omega, \qquad u = 0 \quad \text{on } \Gamma, \tag{5.5}$$

the general WRM statement reads

$$\int_\Omega W^i (\nabla^2 N^j - f)\, d\Omega\, \hat{u}_j = 0. \tag{5.6}$$

Observe that in order to evaluate this integral:

- N^j should have defined second derivatives, i.e. they should be C^1-continuous across elements; and
- W^i can include δ-functions.

Integration by parts results in

$$-\int_\Omega \nabla W^i \cdot \nabla N^j\, d\Omega\, \hat{u}_j - \int_\Omega W^i f\, d\Omega = 0. \tag{5.7}$$

Observe that in this case:

- the order of the maximum derivative has been reduced, implying that a wider space of trial functions can be used;
- the N^j should have defined first derivatives, i.e. they can now be C^0-continuous across elements; and
- the W^i can no longer include δ-functions.

The allowance of C^0-continuous functions is particularly beneficial, as the construction of C^1-continuous functions tends to be cumbersome.

5.2.1. MINIMIZATION PROBLEM

As before with the approximation problem, one may derive the resulting Galerkin WRM from the minimization of a functional. Consider the Rayleigh–Ritz functional

$$I_{rr} = \int_\Omega [\nabla \epsilon^h]^2\, d\Omega = \int_\Omega [\nabla (u^h - u)]^2\, d\Omega \to \text{min}. \tag{5.8}$$

Minimization of this functional is achieved by varying I_{rr} with respect to the available degrees of freedom,

$$\delta I_{rr} = \delta \hat{u}_i \int_\Omega \nabla N^i \cdot (\nabla N^j \hat{u}_j - \nabla u)\, d\Omega = 0. \tag{5.9}$$

All integrals containing u may be eliminated as

$$-\int_\Omega \nabla N^i \cdot \nabla u\, d\Omega = \int_\Omega N^i \nabla^2 u\, d\Omega = \int_\Omega N^i f\, d\Omega, \tag{5.10}$$

resulting in

$$\delta I_{rr} = \delta \hat{u}_i \left(\int_\Omega \nabla N^i \cdot \nabla N^j\, d\Omega\, \hat{u}_j - \int_\Omega N^i f\, d\Omega = 0 \right). \tag{5.11}$$

But this is the same as the Galerkin WRM integral of (5.5)! This implies that the Galerkin choice of taking W^i from the same set as N^i is optimal if the norm given by I_{rr} is used as a measure. We note that I_{rr} is indeed a very good measure, as it is directly related to the energy of the system.

5.2.2. AN EXAMPLE

As a detailed example, consider the regular triangular mesh shown in Figure 5.1. The aim is to assemble all element contributions in order to produce the equation for a typical interior point for the Poisson operator

$$-\nabla^2 u = f. \tag{5.12}$$

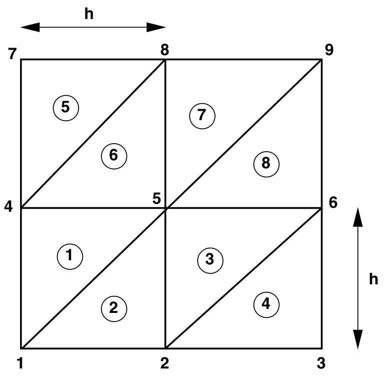


Figure 5.1. Example for Poisson operator

The element connectivity data is given in Table 5.1.

Table 5.1. Element connectivity data `inpoel`

Element	Node 1	Node 2	Node 3
1	1	5	4
2	2	5	1
3	2	6	5
4	2	3	6
5	4	8	7
6	4	5	8
7	5	9	8
8	5	6	9

The possible matrix entries that arise due to the topology of the mesh and the numbering of the nodes are shown in Figure 5.2.

We now proceed to evaluate the individual element matrices. Given the particular mesh under consideration, only two types of elements have to be considered.

5.2.2.1. Element 1

Shape-function derivatives:

$$\begin{bmatrix} N^A \\ N^B \\ N^C \end{bmatrix}_{,x} = \frac{1}{h^2}\begin{bmatrix} 0 \\ h \\ -h \end{bmatrix}, \quad \begin{bmatrix} N^A \\ N^B \\ N^C \end{bmatrix}_{,y} = \frac{1}{h^2}\begin{bmatrix} -h \\ 0 \\ h \end{bmatrix}.$$

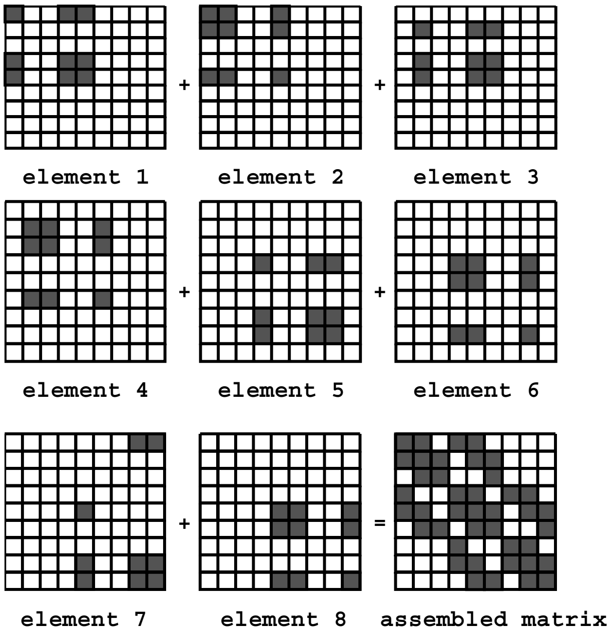


Figure 5.2. Matrix entries

Left-hand side (LHS) contribution:

$$\mathbf{K}_1 \cdot \mathbf{u}_1 = \frac{1}{2h^2}\begin{bmatrix} h^2 & 0 & -h^2 \\ 0 & h^2 & -h^2 \\ -h^2 & -h^2 & 2h^2 \end{bmatrix} \cdot \begin{bmatrix} \hat{u}_1 \\ \hat{u}_5 \\ \hat{u}_4 \end{bmatrix} = \frac{1}{2}\begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & -1 \\ -1 & -1 & 2 \end{bmatrix} \cdot \begin{bmatrix} \hat{u}_1 \\ \hat{u}_5 \\ \hat{u}_4 \end{bmatrix}.$$

Right-hand side (RHS) contribution:

$$\mathbf{M}^1 \cdot \mathbf{f}_1 = \frac{h^2}{24}\begin{bmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{bmatrix} \cdot \begin{bmatrix} \hat{f}_1 \\ \hat{f}_5 \\ \hat{f}_4 \end{bmatrix} = \frac{h^2}{24}\begin{bmatrix} 2\hat{f}_1 + \hat{f}_5 + \hat{f}_4 \\ \hat{f}_1 + 2\hat{f}_5 + \hat{f}_4 \\ \hat{f}_1 + \hat{f}_5 + 2\hat{f}_4 \end{bmatrix}.$$

5.2.2.2. Element 2

Shape-function derivatives:

$$\begin{bmatrix} N^A \\ N^B \\ N^C \end{bmatrix}_{,x} = \frac{1}{h^2}\begin{bmatrix} -h \\ h \\ 0 \end{bmatrix}, \quad \begin{bmatrix} N^A \\ N^B \\ N^C \end{bmatrix}_{,y} = \frac{1}{h^2}\begin{bmatrix} 0 \\ -h \\ h \end{bmatrix}.$$

LHS contribution:

$$\mathbf{K}_2 \cdot \mathbf{u}_2 = \frac{1}{2h^2}\begin{bmatrix} h^2 & -h^2 & 0 \\ -h^2 & 2h^2 & -h^2 \\ 0 & -h^2 & h^2 \end{bmatrix} \cdot \begin{bmatrix} \hat{u}_1 \\ \hat{u}_2 \\ \hat{u}_5 \end{bmatrix} = \frac{1}{2}\begin{bmatrix} 1 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} \hat{u}_1 \\ \hat{u}_2 \\ \hat{u}_5 \end{bmatrix}.$$

RHS contribution:

$$\mathbf{M}^1 \cdot \mathbf{f}_1 = \frac{h^2}{24} \begin{bmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{bmatrix} \cdot \begin{bmatrix} \hat{f}_1 \\ \hat{f}_2 \\ \hat{f}_5 \end{bmatrix} = \frac{h^2}{24} \begin{bmatrix} 2\hat{f}_1 + \hat{f}_2 + \hat{f}_5 \\ \hat{f}_1 + 2\hat{f}_2 + \hat{f}_5 \\ \hat{f}_1 + \hat{f}_2 + 2\hat{f}_5 \end{bmatrix}.$$

The fully assembled form of the equations for point 5 results in

$$4\hat{u}_5 - \hat{u}_2 - \hat{u}_4 - \hat{u}_6 - \hat{u}_8 = \frac{h^2}{12}(6\hat{f}_5 + \hat{f}_1 + \hat{f}_2 + \hat{f}_6 + \hat{f}_4 + \hat{f}_8 + \hat{f}_9). \tag{5.13}$$

This may be compared with a finite difference expansion for point 5:

$$[-\nabla^2 u - f]_{\text{node } 5} = 0, \tag{5.14}$$

$$4\hat{u}_5 - \hat{u}_2 - \hat{u}_4 - \hat{u}_6 - \hat{u}_8 = h^2 \hat{f}_5. \tag{5.15}$$

Observe that the LHS is identical, implying that for the finite element approximation all matrix entries from the ‘diagonal edges’ 1, 5 and 5, 9 are zero. This is the result of the right angle in the elements, which leads to shape functions whose gradients are orthogonal to each other ($\nabla N^i \cdot \nabla N^j = 0$). However, the RHSs are different. The GFEM ‘spreads’ the force function f more evenly over the elements, whereas the finite difference approximation ‘lumps’ f into point-wise contributions. Although the weights of the neighbour points are considerably smaller than the weight of the central node, an asymmetry does occur, as points 3 and 7 are not considered.

5.2.3. TUTORIAL: CODE FRAGMENT FOR HEAT EQUATION

In order to acquaint the newcomer with finite element or finite volume techniques, the explicit coding of the Laplacian RHS will be considered. A more general form of the Laplacian that accounts for spatial variations of material properties is taken as the starting point:

$$\rho c_p T_{,t} = \nabla \cdot \mathbf{k} \cdot \nabla T + S, \tag{5.16}$$

$$T = T_0 \quad \text{on } \Gamma_D,\ q_n := \mathbf{n} \cdot \mathbf{k} \nabla T = q_0 + \alpha(T - T_1) + \beta(T^4 - T_2^4) \quad \text{on } \Gamma_N \tag{5.17}$$

where ρ, c_p, T, $\mathbf{k}$, S, T_0, q_0, α, β, T_1, T_2 denote the density, heat capacitance, temperature, conductivity tensor, sources, prescribed temperature, prescribed fluxes, film coefficient, radiation coefficient and external temperatures, respectively. The Galerkin weighted residual statement

$$\int_\Omega N^j \rho c_p N^i \, d\Omega \, T_{i,t} = -\int_\Omega \nabla N^j \cdot \mathbf{k} \cdot \nabla N^i \, d\Omega \, T_i + \int_\Omega N^j S \, d\Omega + \int_{\Gamma_N} N^j q_n \, d\Gamma \tag{5.18}$$

leads to a *matrix system* for the vector of unknown temperatures $\mathbf{T}$ of the form

$$\mathbf{M} \cdot \mathbf{T}_{,t} = \mathbf{K} \cdot \mathbf{T} + \mathbf{s}. \tag{5.19}$$

All integrals that appear are computed in an element- or face-wise fashion,

$$\int_\Omega \ldots d\Omega = \sum_{el} \int_\Omega \ldots d\Omega_{el}, \quad \int_\Gamma \ldots d\Gamma = \sum_{fa} \int_\Gamma \ldots d\Gamma_{fa}. \tag{5.20}$$

Consider first the domain integrals appearing on the RHS. Assuming linear shape functions N^i, the derivatives of these shape functions are constants, implying that the integrals can be evaluated analytically.

Storing in:

- `geome(1:ndimn*nnode, 1:nelem)` the shape-function derivatives of each element,
- `geome(ndimn*nnode+1,1:nelem)` the volume of each element,
- `diffu(ndimn*ndimn,1:nelem)` the diffusivity tensor in each element, and
- `tempp(1:npoin)` the temperature at each point,

the RHS corresponding to the first domain integral on the RHS may be evaluated as follows for a 2-D application:

```
rhspo(1:npoin)=0                                  ! Initialize rhspo

do ielem=1,nelem                                  ! Loop over the elements
   ipoi1=inpoel(1,ielem)                          ! Nodes of the element
   ipoi2=inpoel(2,ielem)
   ipoi3=inpoel(3,ielem)
   temp1=tempp(ipoi1)                             ! Temperature at the points
   temp2=tempp(ipoi2)
   temp3=tempp(ipoi3)
   rn1x =geome(1,ielem)                           ! Shape-function derivatives
   rn1y =geome(2,ielem)
   rn2x =geome(3,ielem)
   rn2y =geome(4,ielem)
   rn3x =geome(5,ielem)
   rn3y =geome(6,ielem)
   volel=geome(7,ielem)                           ! Volume of the element
   ! Derivatives of the temperature
   tx   =rn1x*temp1+rn2x*temp2+rn3x*temp3
   ty   =rn1y*temp1+rn2y*temp2+rn3y*temp3
   ! Heat fluxes
   fluxx=diffu(1,ielem)*tx+diffu(2,ielem)*ty
   fluxy=diffu(3,ielem)*tx+diffu(4,ielem)*ty
   ! Element RHS
   rele1=volel*(rn1x*fluxx+rn1y*fluxy)
   rele2=volel*(rn2x*fluxx+rn2y*fluxy)
   rele3=volel*(rn3x*fluxx+rn3y*fluxy)
   ! Add element RHS to rhspo
   rhspo(ipoi1)=rhspo(ipoi1)+rele1
   rhspo(ipoi2)=rhspo(ipoi2)+rele2
   rhspo(ipoi3)=rhspo(ipoi3)+rele3
enddo
```

Consider next the second domain integral appearing on the RHS. This is an integral involving a source term that is typically user-defined. Assuming a constant source `souel(1:nelem)` in each element, the RHS is evaluated as follows for a 2-D application:

```
rhspo(1:npoin)=0                              ! Initialize rhspo
cnode=1/float(nnode)                          ! Geometric factor
do ielem=1,nelem                              ! Loop over the elements
  ipoi1=inpoel(1,ielem)                       ! Nodes of the element
  ipoi2=inpoel(2,ielem)
  ipoi3=inpoel(3,ielem)
  rhsel=cnode*souel(ielem)*geome(7,ielem)
  ! Add element RHS to rhspo
  rhspo(ipoi1)=rhspo(ipoi1)+rhsel
  rhspo(ipoi2)=rhspo(ipoi2)+rhsel
  rhspo(ipoi3)=rhspo(ipoi3)+rhsel
enddo
```

5.3. Recovery of derivatives

A recurring task for flow solvers, error indicators, visualization and other areas of CFD is the evaluation of derivatives at points or in the elements. While evaluating a first-order derivative in elements is an easy matter, the direct evaluation at points is impossible, as the shape functions exhibit a discontinuity in slope there. A typical example is the linear shape function (tent function) displayed in Figure 5.3.

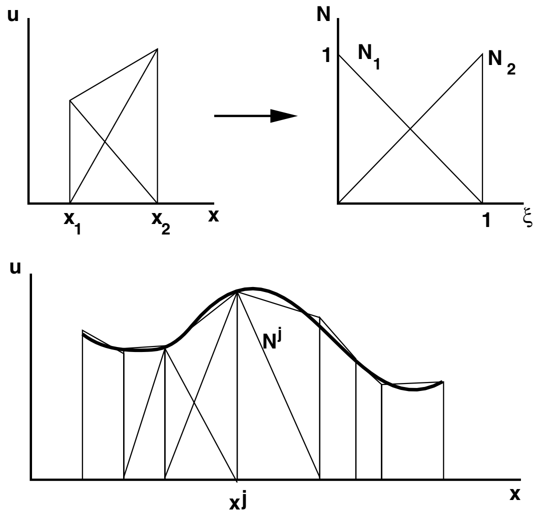


Figure 5.3. Linear shape function

If a derivative of the unknowns is desired at points, it must be obtained using so-called ‘recovery’ procedures employing WRMs.

5.3.1. FIRST DERIVATIVES

The recovery of any derivative starts with the assumption that the derivative sought may also be expressed via shape functions $\tilde{N}^i$,

$$\frac{\partial u}{\partial s} \approx \tilde{N}^i \hat{u}_i'. \tag{5.21}$$

Here $\hat{u}_i'$ denotes the value of the first-order derivative of u with respect to s at the location of point $\mathbf{x}_i$. The direction s is arbitrary, i.e. could correspond to x, y, z, or any other direction. The original assumption for the unknown function u was

$$u \approx N^i \hat{u}_i, \tag{5.22}$$

implying

$$\frac{\partial u}{\partial s} \approx \frac{\partial N^i}{\partial s} \hat{u}_i. \tag{5.23}$$

Comparing (5.21) and (5.23), we have

$$\tilde{N}^i \hat{u}_i' \approx \frac{\partial N^k}{\partial s} \hat{u}_k. \tag{5.24}$$

Weighting this relation with shape-functions W^i, we obtain

$$\int_\Omega W^i \tilde{N}^j \, d\Omega \, \hat{u}_j' = \int_\Omega W^i \frac{\partial N^j}{\partial s} \, d\Omega \, \hat{u}_j. \tag{5.25}$$

For the special (but common) case $W^i = \tilde{N}^i = N^i$, which corresponds to a Galerkin WRM, the recovery reduces to

$$\mathbf{M}_c \mathbf{u}' = \int_\Omega N^i N^j \, d\Omega \, \hat{u}_j' = \int_\Omega N^i \frac{\partial N^j}{\partial s} \, d\Omega \, \hat{u}_j. \tag{5.26}$$

Observe that on the LHS the mass matrix is obtained. If the inversion of this matrix seems too costly, the lumped mass matrix $\mathbf{M}_l$ may be employed instead.

5.3.2. SECOND DERIVATIVES

For second derivatives, two possibilities appear:

(a) evaluation of first derivatives, followed by evaluation of first derivatives of first derivatives (i.e. a recursive two-pass strategy); or

(b) direct evaluation of second derivatives via integration by parts (i.e. a one-pass strategy).

The second strategy is faster, and is therefore employed more often. As before, one may start with the assumption that

$$\frac{\partial^2 u}{\partial s^2} u \approx \tilde{N}^i \hat{u}_i''. \tag{5.27}$$

Applying the same steps as described above, the weighted residual statement results in

$$\int_\Omega W^i \tilde{N}^j \, d\Omega \, \hat{u}''_j = \int_\Omega W^i \frac{\partial^2 N^j}{\partial s^2} \, d\Omega \, \hat{u}_j. \tag{5.28}$$

The difficulty of evaluating second derivatives for the shape functions is circumvented via integration by parts:

$$\int_\Omega W^i \tilde{N}^j \, d\Omega \, \hat{u}''_j = -\int_\Omega \frac{\partial W^i}{\partial s} \frac{\partial N^j}{\partial s} \, d\Omega \, \hat{u}_j + \int_\Gamma W^i n_s \frac{\partial N^j}{\partial s} \, d\Gamma \, \hat{u}_j. \tag{5.29}$$

For the special (but common) case $W^i = \tilde{N}^i = N^i$, which corresponds to a Galerkin WRM, the recovery of a second-order derivative reduces to

$$\mathbf{M}_c \mathbf{u}'' = \int_\Omega N^i N^j \, d\Omega \, \hat{u}''_j = -\int_\Omega \frac{\partial N^i}{\partial s} \frac{\partial N^j}{\partial s} \, d\Omega \, \hat{u}_j + \int_\Gamma N^i n_s \frac{\partial N^j}{\partial s} \, d\Gamma \, \hat{u}_j. \tag{5.30}$$

As before, on the LHS the mass matrix is obtained.

5.3.3. HIGHER DERIVATIVES

While integration by parts was sufficient for second derivatives, higher derivatives require a recursive evaluation. For an even derivative of order $p = 2n$, the second derivatives are evaluated first. Using these values, fourth derivatives are computed, then sixth derivatives, etc. For an uneven derivative of order $p = 2n + 1$, n second derivative evaluations are mixed with a first derivative evaluation.

6 DISCRETIZATION IN TIME

In the previous chapter the spatial discretization of operators was considered. We now turn our attention to temporal discretizations. We could operate as before, and treat the temporal dimension as just another spatial dimension. This is possible, and has been considered in the past (Zienkiewicz and Taylor (1988)). There are three main reasons why this approach has not found widespread acceptance.

(a) For higher-order schemes, the tight coupling of several timesteps tends to produce exceedingly large matrix systems.

(b) For lower-order schemes, the resulting algorithms are the same as finite difference schemes. As these are easier to derive, and more man-hours have been devoted to their study, it seems advantageous to employ them in this context.

(c) Time, unlike space, has a definite direction. Therefore, schemes that reflect this hyperbolic character will be the most appropriate. Finite difference or low-order finite elements in time do reflect this character correctly.

In what follows, we will assume that we have already accomplished the spatial discretization of the operator. Therefore, the problem to be solved may be stated as a system of nonlinear ordinary differential equations (ODEs) of the form

$$\mathbf{u}_{,t} = \mathbf{r}(t, \mathbf{u}). \tag{6.1}$$

Timestepping schemes may be divided into explicit and implicit schemes.

6.1. Explicit schemes

Explicit schemes take the RHS vector $\mathbf{r}$ at a known time (or at several known times), and predict the unknowns $\mathbf{u}$ at some time in the future based on it. The simplest such case is the forward Euler scheme, given by

$$\Delta\mathbf{u}^{n+1} = \mathbf{u}^{n+1} - \mathbf{u}^n = \Delta t\mathbf{r}(t^n, \mathbf{u}^n). \tag{6.2}$$

An immediate generalization to higher-order schemes is given by the family of explicit *Runge–Kutta* (RK) methods, which may be expressed as

$$\mathbf{u}^{n+1} = \mathbf{u}^n + \Delta t b_i \mathbf{r}^i, \tag{6.3}$$

$$\mathbf{r}^i = \mathbf{r}(t^n + c_i\Delta t, \mathbf{u}^n + \Delta t a_{ij}\mathbf{r}^j), \quad i = 1, s,\ j = 1, s-1. \tag{6.4}$$

Any particular RK method is defined by the number of stages s and the coefficients a_{ij}, $1 \leq j < i \leq s$, b_i, $i = 1, s$ and c_i, $i = 2, s$. These coefficients are usually arranged in a table known as a Butcher tableau (see Butcher (2003)):

	$\mathbf{r}^1$	$\mathbf{r}^2$	.	.	$\mathbf{r}^{s-1}$	$\mathbf{r}^s$
0						
c_2	a_{21}					
c_3	a_{31}	a_{32}				
.	.		.			
.	.			.		
c_s	a_{s1}	a_{s2}	.	.	$a_{s,s-1}$	
	b_1	b_2	.	.	b_{s-1}	b_s

The classic fourth-order RK scheme is given by:

	$\mathbf{r}^1$	$\mathbf{r}^2$	$\mathbf{r}^3$	$\mathbf{r}^4$
0				
1/2	1/2			
1/2	0	1/2		
1	0	0	1	
	1/6	1/3	1/3	1/6

Observe that schemes of this kind require the storage of *several* copies of the unknowns/RHSs, as the final result requires $\mathbf{r}^i$, $i = 1, s$. For this reason, so-called minimal storage RK schemes that only require *one* extra copy of the unknowns/RHSs have been extensively used in CFD. These schemes may be obtained by deriving consistent coefficients for RK schemes where $a_{ij} = 0$, $\forall j < i - 1$, and $b_i = 0$, $\forall i < s$. These minimal-storage RK schemes for Euler and Navier–Stokes solvers have been studied extensively over the past two decades. The main impetus for this focus was the paper of Jameson *et al.* (1981), which set the stage for a number of popular CFD solvers. An s-stage minimal storage RK scheme may be recast into the form

$$\Delta\mathbf{u}^{n+i} = \alpha_i \Delta t \mathbf{r}(\mathbf{u}^n + \Delta\mathbf{u}^{n+i-1}), \quad i = 1, s, \quad \Delta\mathbf{u}^0 = 0. \tag{6.5}$$

The coefficients α_i are chosen according to desired properties, such as damping (e.g. for multigrid smoothing) and temporal order of accuracy. Common choices are:

(a) one-stage scheme (forward Euler): $\alpha_1 = 1.0$;

(b) two-stage scheme: $\alpha_1 = 0.5$, $\alpha_2 = 1$;

(c) three-stage scheme: $\alpha_1 = 0.6$, $\alpha_2 = 0.6$, $\alpha_3 = 1$;

(d) four-stage scheme for steady-state, one-grid: $\alpha_1 = 1/4$, $\alpha_2 = 1/3$, $\alpha_3 = 1/2$, $\alpha_4 = 1$;

(e) four-stage scheme for multigrid: $\alpha_1 = 1/4$, $\alpha_2 = 1/2$, $\alpha_3 = 0.5$, $\alpha_4 = 1$.

Note that for linear ODEs the choice

$$\alpha_i = \frac{1}{s + 1 - i}, \quad i = 1, s \tag{6.6}$$

yields a scheme that is of sth-order accuracy in time (!).

The main properties of explicit schemes are:

- they allow for an arbitrary order of temporal accuracy;

- they are easy to code;
- the enforcement of boundary conditions is simple;
- vectorization or parallelization is straightforward;
- they are easy to maintain/upgrade;
- the allowable timestep Δt is limited by stability constraints, such as the so-called Courant–Friedrichs–Levy (CFL) number, which for a hyperbolic system of PDEs is given by the following relation of timestep Δt, size of the element h and maximum eigenvalue $\lambda_{\max}$:
$$\mathrm{CFL} = \frac{\Delta t \lambda_{\max}}{h}. \tag{6.7}$$

6.2. Implicit schemes

Given that in many applications the time scales required for accuracy allow timesteps that are much larger than the ones permitted for explicit schemes, implicit schemes have been pursued for over three decades. The simplest of these schemes use a RHS that is evaluated somewhere between the present time position t^n and the next time position t^{n+1}:

$$\Delta \mathbf{u} = \mathbf{u}^{n+1} - \mathbf{u}^n = \Delta t \mathbf{r}(\mathbf{u}^{n+\Theta}). \tag{6.8}$$

The approximation most often used linearizes the RHS using the Jacobian $\mathbf{A}^n$:

$$\mathbf{r}^{n+\Theta} = \mathbf{r}^n + \left.\frac{\partial \mathbf{r}}{\partial \mathbf{u}}\right|^n \cdot \Theta\Delta\mathbf{u} = \mathbf{r}^n + \mathbf{A}^n \cdot \Theta\Delta\mathbf{u}. \tag{6.9}$$

Equation (6.8) may now be recast in the final form

$$[1 - \Delta t \Theta \mathbf{A}^n] \cdot \Delta\mathbf{u} = \mathbf{r}^n. \tag{6.10}$$

Popular choices for Θ are:
$\Theta = 1.0$: backward Euler (first-order accurate);
$\Theta = 0.5$: Crank-Nicholson (second-order accurate).

The generalization of these one-step schemes to so-called linear multistep schemes is given by
$$\alpha_j \mathbf{u}^{n+j} = \Delta t \beta_j \mathbf{r}(\mathbf{u}^{n+j}), \quad j = 0, k. \tag{6.11}$$

However, the hope of achieving high-order implicit schemes via this generalization are rendered futile by Dahlquist’s (1963) theorem that states that there exists no unconditionally stable linear multistep scheme that is of order higher than two. For this reason, schemes of this kind have not been used extensively. A possible way to circumvent Dahlquist’s theorem is via implicit RK methods (Butcher (1964), Hairer and Wanner (1981), Cash (2003)) which have recently been studied for CFD applications (Bijl *et al.* (2002), Jothiprasad *et al.* (2003)).

The main properties of implicit schemes are:

- the maximum order of accuracy for unconditionally stable linear multistep schemes is two;

- one may take ‘arbitrary’ timesteps Δt, i.e. Δt is governed only by accuracy considerations and not stability;
- the timestep Δt is independent of the grid, implying that one may take reasonable timesteps even for distorted grids;
- implicit schemes have an overhead due to the required solution of the large system of equations appearing on the LHS of (6.10).

6.2.1. SITUATIONS WHERE IMPLICIT SCHEMES PAY OFF

Although implicit schemes would appear to be much more expensive and difficult to implement and maintain than explicit schemes, there are certain classes of problems where they pay off. These are as follows.

(a) *Severe physical stiffness*. In this case, we have

$$\Delta t|_{\text{phys. relevant}} \gg \Delta t_{\text{CFL}}.$$

Examples where this may happen are: speed of sound limitations for boundary layers in low Mach-number or incompressible flows, heat conduction, etc.

(b) *Severe mesh stiffness*. This may be due to small elements, distorted elements, geometrically difficult surfaces, deficient grid generators, etc.

6.3. A word of caution

Before going on, it seems prudent to remind the newcomer that the claim of ‘arbitrary timesteps’ is seldomly realized in practice. Most of the interesting engineering problems are highly nonlinear (otherwise they would not be interesting), implying that any linearization during stability analysis may yield incorrect estimates. Most implicit CFD codes will not run with CFL numbers that are well above 100. In fact, the highest rates of convergence to the steady state are usually attained at CFL $= O(10)$. Moreover, even if the scheme is stable, the choice of too large a timestep Δt for systems of nonlinear PDEs may lead to unphysical, chaotic solutions (Yee *et al.* (1991), Yee and Sweby (1994), Yee (2001)). These are solutions that are unsteady, remain stable, look perfectly plausible and yet are purely an artifact of the large timesteps employed. The reverse has also been reported: one may achieve a steady solution that is an artifact of the large timestep. As the timestep is diminished, the correct unsteady solution is retrieved. To complicate matters further, the possibility of the onset of ‘numerical chaos’ happens to occur for timestep values that are close to the explicit stability limit CFL $= O(1)$. Thus, one should always conduct a convergence study or carry out an approximate error analysis when running with high CFL numbers.

7 SOLUTION OF LARGE SYSTEMS OF EQUATIONS

As we saw from the preceeding sections, both the straightforward spatial discretization of a steady-state problem and the implicit time discretization of a transient problem will yield a large system of coupled equations of the form

$$\mathbf{K} \cdot \mathbf{u} = \mathbf{f}. \tag{7.1}$$

There are two basic approaches to the solution of this problem:

(a) directly, by some form of Gaussian elimination; or

(b) iteratively.

We will consider both here, as any solver requires one of these two, if not both.

7.1. Direct solvers

The rapid increase in computer memory, and their suitability for shared-memory multi-processing computing environments, has lead to a revival of direct solvers (see, e.g., Giles *et al.* (1985)). Three-dimensional problems once considered unmanageable due to their size are now being solved routinely by direct solvers (Wigton *et al.* (1985), Nguyen *et al.* (1990), Dutto *et al.* (1994), Luo *et al.* (1994c)). This section reviews the direct solvers most commonly used.

7.1.1. GAUSSIAN ELIMINATION

This is the classic direct solver. The idea is to add (subtract) appropriately scaled rows in the system of equations in order to arrive at an upper triangular matrix (see Figure 7.1(a)):

$$\mathbf{K} \cdot \mathbf{u} = \mathbf{f} \rightarrow \mathbf{U} \cdot \mathbf{u} = \mathbf{f}'. \tag{7.2}$$

To see how this is done in more detail, and to obtain an estimate of the work involved, we rewrite (7.1) as

$$K^{ij} u_j = f^i. \tag{7.3}$$

Suppose that the objective is to obtain vanishing entries for all matrix elements located in the jth column below the diagonal K^{jj} entry. This can be achieved by adding to the kth row ($k > j$) an appropriate fraction of the jth row, resulting in

$$(K^{kl} + \alpha_k K^{jl}) u_l = f^k + \alpha_k f^j, \quad k > j. \tag{7.4}$$

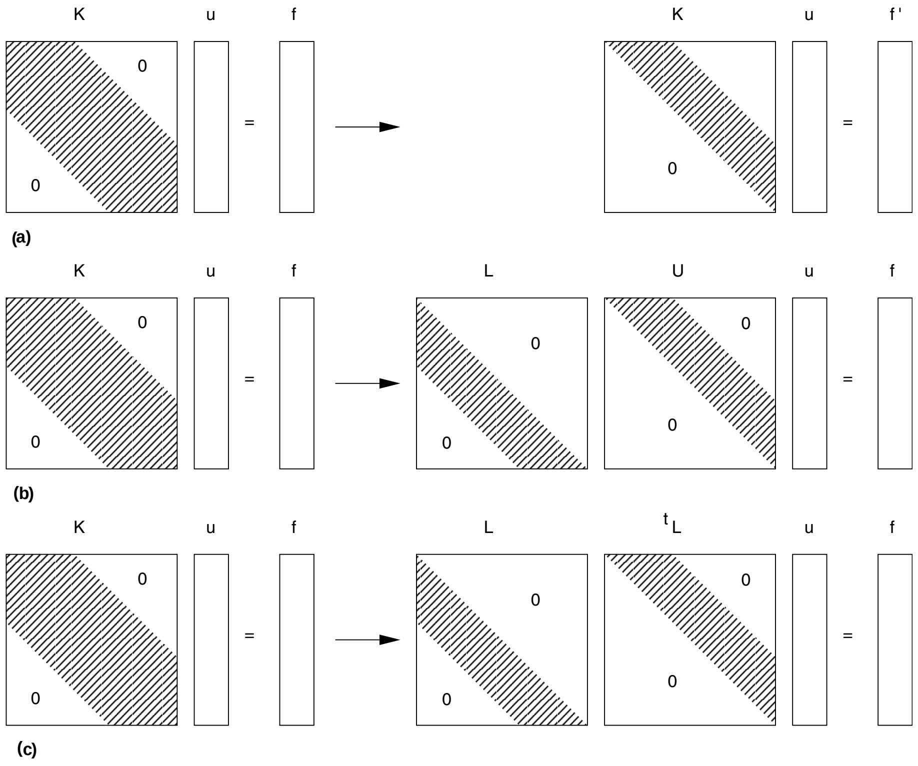


Figure 7.1. Direct solvers: (a) Gaussian elimination; (b) Crout decomposition; (c) Cholesky

Such an addition of rows will not change the final result for **u** and is therefore allowable. For the elements located in the jth column below the diagonal K^{jj} entry to vanish, we must have

$$\alpha_k = -\frac{K^{kj}}{K^{jj}}. \tag{7.5}$$

The process is started with the first column and repeated for all remaining ones. Once an upper triangular form has been obtained, the solution becomes trivial. Starting from the bottom right entry, all unknowns **u** are obtained recursively from the preceding column:

$$u^i = (U^{ii})^{-1}(f'^i - U^{ij})u^j, \quad j > i. \tag{7.6}$$

The work required to solve a system of equations in this way is as follows. For a full matrix, the matrix triangularization requires $O(N)$ multiplications for each column, i.e. $O(N^2)$ operations for all columns. As this has to be repeated for each row, the total estimate is $O(N^3)$. The solution phase requires $O(N)$ operations for each row, i.e. $O(N^2)$ operations for all unknowns. If the matrix has a banded structure with bandwidth N_{ba}, these estimates reduce to $O(NN_{ba}^2)$ for the matrix triangularization and $O(NN_{ba})$ for the solution phase. Gaussian elimination is seldomly used in practice, as the transformation of the matrix changes the RHS vector, thereby rendering it inefficient for systems with multiple RHS.

7.1.2. CROUT ELIMINATION

In this case, the matrix is decomposed into an upper and lower triangular portion (see Figure 7.1(b)):

$$\mathbf{K} \cdot \mathbf{u} = \mathbf{L} \cdot \mathbf{U} \cdot \mathbf{u} = \mathbf{f}. \tag{7.7}$$

One can choose the diagonal entries of either the upper or the lower triangular matrix to be of unit value. Suppose that the diagonal entries of the upper triangular matrix are so chosen, and that the matrix has been decomposed up to entry $i-1, i-1$, i.e. the entries $1:i-1; 1:-1$ of $\mathbf{L}$ and $\mathbf{U}$ are known. The entries along row i are given by

$$K^{ij} = \sum_{k=1}^{j} L^{ik} U^{kj}. \tag{7.8}$$

Given that $U^{kk} = 1$, we obtain, recursively,

$$L^{ij} = K^{ij} - \sum_{k=1}^{j-1} L^{ik} U^{kj}. \tag{7.9}$$

The entries along column i are given by

$$K^{ji} = \sum_{k=1}^{j} L^{jk} U^{ki}. \tag{7.10}$$

This allows the recursive calculation of U^{ji} from

$$U^{ji} = \frac{1}{L^{jj}} \left[K^{ji} - \sum_{k=1}^{j-1} L^{jk} U^{ki} \right]. \tag{7.11}$$

The value for the diagonal entry L^{ii} is then obtained from

$$L^{ii} = K^{ii} - \sum_{k=1}^{i-1} L^{ik} U^{ki}. \tag{7.12}$$

This completes the decomposition of the ith row and column. The process is started with the first row and column, and repeated for all remaining ones. Once the decomposition is complete, the system is solved in two steps:

- Forward substitution: $\mathbf{L} \cdot \mathbf{v} = \mathbf{f}$, followed by

- Backward substitution: $\mathbf{U} \cdot \mathbf{u} = \mathbf{v}$.

Observe that the RHS is not affected by the decomposition process. This allows the simple solution of multiple RHS.

7.1.3. CHOLESKY ELIMINATION

This special decomposition is only applicable to symmetric matrices. The algorithm is almost the same as the Crout decomposition, except that square roots are taken for the diagonal elements. This seemingly innocuous change has a very beneficial effect on rounding errors (Zurmühl (1964)).

All direct solvers have a storage and operation count as follows:

Operations: $O(N_{eq}N_{ba}^2)$,
Storage: $O(N_{eq}N_{ba})$,

where N_{ba} is the bandwidth of the matrix. As the bandwidth increases, so do the possibilities for vectorization and parallelization. Very efficient direct solvers for multi-processor vector machines have been reported (Nguyen *et al.* (1990), Dutto *et al.* (1994)). As the number of equations N_{eq} is fixed, the most important issue when trying to optimize direct solvers is the reduction of the bandwidth N_{ba}. This is an optimization problem that is Np complete, i.e. many heuristic solutions can be obtained that give the same or nearly the same cost function (bandwidth in this case), but the optimum solution is practically impossible to obtain. Moreover, the optimum solution may not be unique. As an example, consider a square domain discretized by $N \times N$ quadrilateral elements. Suppose further that Poisson's equation with Dirichlet boundary conditions is to be solved numerically, and that the spatial discretization consists of bilinear finite elements. Starting any numbering in the same way from each of the four corners will give the same bandwidth, storage and CPU requirements, and hence the same cost function. Bandwidth reduction implies a renumbering of the nodes, with the aim of bringing all matrix entries closer to the diagonal. The main techniques used to accomplish this are (Piessanetzky (1984)):

- Cuthill–McKee (CMK), and reverse CMK, which order the points according to lowest connectivity with surrounding points at each level of the corresponding graph (Cuthill and McKee (1969));
- wavefront, whereby the mesh is renumbered according to an advancing front; and
- nested dissection, where the argument of bandwidth reduction due to recursive subdivision of domains is employed (George and Liu (1981)).

The first two approaches have been used extensively in structural finite element analysis. Many variations have been reported, particularly for the 'non-exact' parameters such as starting point, depth of search and trees, data structures, etc. Renumbering strategies reappear when trying to minimize cache-misses, and are considered in more depth in Chapter 15.

7.2. Iterative solvers

When (7.1) is solved iteratively, the matrix $\mathbf{K}$ is not inverted directly, but the original problem is replaced by a sequence of solutions of the form

$$\tilde{\mathbf{K}} \cdot (\mathbf{u}^{n+1} - \mathbf{u}^n) = \tilde{\mathbf{K}} \cdot \Delta\mathbf{u} = \Delta\tau\mathbf{r} = \Delta\tau(\mathbf{f} - \mathbf{K} \cdot \mathbf{u}). \tag{7.13}$$

The vector $\mathbf{r}$ is called the residual vector, and $\tilde{\mathbf{K}}$ the *preconditioning matrix*. The case $\tilde{\mathbf{K}} = \mathbf{K}$ corresponds to a direct solution, and the sequence of solutions stops after one iteration. The aim is to approximate $\mathbf{K}$ by some low-cost, yet 'good' $\tilde{\mathbf{K}}$. 'Good' in this context means that:

(a) $\tilde{\mathbf{K}}$ is inexpensive to decompose or solve for;

(b) $\tilde{\mathbf{K}}$ contains relevant information (eigenvalues, eigenvectors) about $\mathbf{K}$.

Unfortunately, these requirements are contradictory. What tends to happen is that the low-eigenvalue (i.e. long-wavelength eigenmodes) information is lost when $\mathbf{K}$ is approximated with $\tilde{\mathbf{K}}$. To circumvent this deficiency, most practical iterative solvers employ a 'globalization' procedure that counteracts this loss of low-eigenvalue information. Both the approximation and the globalization algorithms employed may be grouped into three families: operator-based, grid-based and matrix-based. We just mention some examples here.

Formation of $\tilde{\mathbf{K}}$:

F1. Operator-based: approximate factorization of implicit FDM CFD codes (Briley and McDonald (1977), Beam and Warming (1978), MacCormack (1982));

F2. Grid-based: point/ element-by-element/ red–black/ line/ zebra/ snake/ linelet/ plane relaxation/ Jacobi/ Gauss–Seidel/ etc. (Wesseling (2004));

F3. Matrix-based: incomplete lower-upper (LU), drop-tolerances, etc. (Nicolaides (1987), Saad (2003), van der Vorst (2003)).

Globalization or acceleration:

G1. Operator-based: Tshebishev, supersteps, etc.;

G2. Grid-based: projection (one coarser grid), multigrid (n coarser grids);

G3. Matrix-based: dominant eigenvalue extrapolation, conjugate gradient (CG), generalized minimal residuals (GMRES), algebraic multigrid (AMG).

7.2.1. MATRIX PRECONDITIONING

In order to be more specific about the different techniques, we rewrite the matrix $\mathbf{K}$ either as a sum of lower, diagonal and upper parts

$$\mathbf{K} = \mathbf{L} + \mathbf{D} + \mathbf{U}, \tag{7.14}$$

or as the product of a number of submatrices

$$\mathbf{K} = \prod_{j=1}^{m} \mathbf{K}_j . \tag{7.15}$$

One can then classify the different preconditioners by the degree of discrepancy (or neglect) between $\mathbf{K}$ and $\tilde{\mathbf{K}}$.

7.2.1.1. Diagonal preconditioning

The simplest preconditioners are obtained by neglecting all off-diagonal matrix entries, resulting in *diagonal preconditioning*

$$\tilde{\mathbf{K}} = \mathbf{D}. \tag{7.16}$$

The physical implication of this simplification is that any transfer of information between points or elements can only be accomplished on the RHS during the iterations (equation (7.13)). This implies that information can only travel one element per iteration, and is similar to explicit timestepping with local timesteps.

7.2.1.2. Block-diagonal preconditioning

For systems of equations it is often necessary to revert to *block-diagonal preconditioning*. All the matrix entries that correspond to edges $i, j, \forall i \neq j$ are still neglected, but the entries for $i, j, \forall i = j$ are kept. The result is a set of blocks of magnitude neqns*neqns along the diagonal. For the Euler equations, this results in 5×5 blocks. For the Navier–Stokes equations with a $k - \epsilon$ turbulence model 7×7 blocks are obtained. As before, the propagation of information between gridpoints can only occur on the RHS during the iterations, at a maximum speed of one element per iteration. The advantage of block-diagonal preconditioning is that it removes the stiffness that may result from equations with different time scales. A typical class of problems for which block-diagonal preconditioning is commonly used is chemically reacting flows, where the time scales of chemical reactions may be orders of magnitude smaller than the (physically interesting) advection time scale of the fluid.

7.2.1.3. LU preconditioning

Although point preconditioners are extremely fast, all the inter-element propagation of information occurs on the RHS, resulting in slow convergence rates. Faster information transfer can only be achieved by neglecting fewer entries from $\mathbf{K}$ in $\tilde{\mathbf{K}}$, albeit at higher CPU and storage costs. If we recall that the solution of a lower (or upper) matrix by itself is simple, a natural way to obtain better preconditioners is to attempt a preconditioner of the form

$$\tilde{\mathbf{K}} = \tilde{\mathbf{K}}^L \cdot \tilde{\mathbf{K}}^U. \tag{7.17}$$

Given a residual $\mathbf{r} = \mathbf{f} - \mathbf{K} \cdot \mathbf{u}$, the new increments are obtained from

$$\tilde{\mathbf{K}}^L \cdot \Delta\mathbf{u}' = \mathbf{r}, \quad \tilde{\mathbf{K}}^U \cdot \Delta\mathbf{u} = \Delta\mathbf{u}'. \tag{7.18}$$

Consider the physical implication of $\tilde{\mathbf{K}}^L$: the unknowns corresponding to point i take into account the new values already obtained for points $i - 1, i - 2, i - 3, \ldots, 1$. Likewise, for $\tilde{\mathbf{K}}^U$, the unknowns corresponding to point i take into account the new values already obtained for points $i + 1, i + 2, i + 3, \ldots, n$. This implies that the information flows with the numbering of the points. In order to propagate the information evenly, the $\tilde{\mathbf{K}}^L$, $\tilde{\mathbf{K}}^U$ are invoked alternately, but in some cases it may be advisable to have more than one numbering for the points to achieve consistent convergence rates. If the numerical flow of information can be matched to the physical flow of information, a very good preconditioner is achieved. Cases where this is possible are supersonic flows, where the numbering of the points matches the streamlines.

Gauss–Seidel variants

The perhaps simplest LU preconditioner is given by the choice

$$\tilde{\mathbf{K}}^L = \mathbf{L} + \mathbf{D}, \quad \tilde{\mathbf{K}}^U = \mathbf{D} + \mathbf{U}. \tag{7.19}$$

The resulting matrix $\tilde{\mathbf{K}}$ is then

$$\tilde{\mathbf{K}} = \tilde{\mathbf{K}}^L \cdot \tilde{\mathbf{K}}^U = (\mathbf{L} + \mathbf{D}) \cdot (\mathbf{D} + \mathbf{U}) = \mathbf{L} \cdot \mathbf{D} + \mathbf{L} \cdot \mathbf{U} + \mathbf{D} \cdot \mathbf{D} + \mathbf{D} \cdot \mathbf{U}. \tag{7.20}$$

Comparing this last expression to (7.14), we see that this form of LU decomposition does not approximate the original matrix $\mathbf{K}$ well. An extra diagonal term has appeared. This may be remedied by interposing the inverse of the diagonal between the lower and upper matrices, resulting in

$$\tilde{\mathbf{K}} = \tilde{\mathbf{K}}^L \cdot \mathbf{D}^{-1} \cdot \tilde{\mathbf{K}}^U = (\mathbf{L} + \mathbf{D}) \cdot \mathbf{D}^{-1} \cdot (\mathbf{D} + \mathbf{U}) = \mathbf{K} + \mathbf{L} \cdot \mathbf{D}^{-1} \cdot \mathbf{U}. \tag{7.21}$$

The error may also be mitigated by adding, for subsequent iterations, a correction with the latest information of the unknowns. This leads to two commonly used schemes:

- Gauss–Seidel (GS)

$$\begin{aligned} (\mathbf{L} + \mathbf{D}) \cdot \Delta\mathbf{u}^1 &= \mathbf{r} - \mathbf{U} \cdot \Delta\mathbf{u}^0, \\ (\mathbf{D} + \mathbf{U}) \cdot \Delta\mathbf{u} &= \mathbf{r} - \mathbf{L} \cdot \Delta\mathbf{u}^1, \end{aligned} \tag{7.22}$$

  which is equivalent to

$$\mathbf{K}\Delta\mathbf{u} = (\mathbf{L} + \mathbf{D} + \mathbf{U}) \cdot \Delta\mathbf{u} = \mathbf{r} + \mathbf{L} \cdot (\Delta\mathbf{u} - \Delta\mathbf{u}^1); \tag{7.23}$$

- lower-upper symmetric GS (LU-SGS)

$$(\mathbf{L} + \mathbf{D}) \cdot \mathbf{D}^{-1} \cdot (\mathbf{D} + \mathbf{U}) \cdot \Delta\mathbf{u} = \mathbf{r} + \mathbf{L} \cdot \mathbf{D}^{-1} \cdot \mathbf{U} \cdot \Delta\mathbf{u}^0,$$

  which is equivalent to

$$\mathbf{K}\Delta\mathbf{u} = (\mathbf{L} + \mathbf{D} + \mathbf{U}) \cdot \Delta\mathbf{u} = \mathbf{r} + \mathbf{L} \cdot \mathbf{D}^{-1} \cdot \mathbf{U} \cdot (\Delta\mathbf{u}^0 - \Delta\mathbf{u}). \tag{7.24}$$

In most cases $\Delta\mathbf{u}^0 = 0$. GS and LU-SGS have been used extensively in CFD, both as solvers and as preconditioners. In this context, very elaborate techniques that combine physical insight, local eigenvalue decompositions and scheme switching have produced very fast and robust preconditioners (Sharov and Nakahashi (1998), Luo *et al.* (1998), Sharov *et al.* (2000a), Luo *et al.* (2001)).

Diagonal+1 preconditioning

Consider a structured grid of $m \times n$ points. Furthermore, assume that a discretization of the Laplacian using the standard stencil

$$-u_{i-1,j} - u_{i,j-1} + 4u_{i,j} - u_{i+1,j} - u_{i,j+1} = r_i \tag{7.25}$$

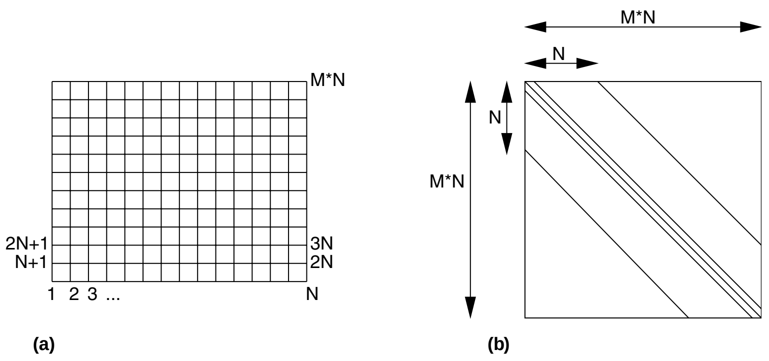


Figure 7.2. Matrix resulting from $m \times n$ structured grid

is being performed. The resulting **K** matrix for the numbering shown in Figure 7.2(a) is depicted in Figure 7.2(b). As one can see, **K** consists of a tridiagonal core $\mathbf{D}'$ and regular off-diagonal ‘bands’.

If **K** is rewritten as

$$\mathbf{K} = \mathbf{L}' + \mathbf{D}' + \mathbf{U}', \tag{7.26}$$

the diagonal+1 preconditioning is defined by

$$\tilde{\mathbf{K}} = \mathbf{D}'. \tag{7.27}$$

In this case, the information will be propagated rapidly via the LHS between all the points that form a tridiagonal system of equations, and slowly on the RHS between all other points. As before, the ordering of the points guides the propagation of information during the iterative procedure. Tridiagonal or block-tridiagonal systems result from point orderings that form ‘lines’ or ‘snakes’ when the points are renumbered. For example, the ordering shown in Figure 7.2 resulted in a considerable number of ‘lines’, leading to an equal number of tridiagonal systems. Given that the fastest information flow is according to the point numbering, the renumbering of points should form ‘lines’ that are in the direction of maximum stiffness. In this way, the highest possible similarity between $\tilde{\mathbf{K}}$ and **K** is achieved (Hassan *et al.* (1990), Martin and Löhner (1992), Mavriplis (1995), Soto *et al.* (2003)). For cases where no discernable spatial direction for stiffness exists, several point renumberings should be employed, with the aim of covering as many $i, j, \forall i \neq j$ entries as possible.

Diagonal+1 Gauss–Seidel

As before, the unknowns already obtained during the solution of $\tilde{\mathbf{K}} = \mathbf{D}'$ can be re-used with minor additional effort, resulting in diagonal+1 GS preconditioning. For structured grids, this type of preconditioning is referred to as line GS relaxation. The resulting preconditioning matrices are of the form

$$\tilde{\mathbf{K}}^L = \mathbf{L}' + \mathbf{D}', \quad \tilde{\mathbf{K}}^U = \mathbf{D}' + \mathbf{U}'. \tag{7.28}$$

7.2.1.4. Incomplete lower-upper preconditioning

All the preconditioners described so far avoided the large operation count and storage requirements of a direct inversion of $\mathbf{K}$ by staying close to the diagonal when operating with $\tilde{\mathbf{K}}$. For incomplete lower-upper (ILU) preconditioning, the product decomposition of the Crout solver

$$\mathbf{K} = \mathbf{L} \cdot \mathbf{U} \tag{7.29}$$

is used, but the fill-in that occurs for all the off-diagonal entries within the band is neglected. This rejection of fill-in can be based on integer logic (i.e. allowing `NFILR` fill-ins per column) or based on some drop-tolerance (i.e. neglecting all entries whose magnitudes are below a threshold). The resulting preconditioning matrix is of the form

$$\tilde{\mathbf{K}} = \tilde{\mathbf{L}} \cdot \tilde{\mathbf{U}} \cdot \tag{7.30}$$

If $\mathbf{K}$ is tridiagonal, then $\tilde{\mathbf{K}} = \mathbf{K}$, implying perfect preconditioning. The observation often made is that the quality of $\tilde{\mathbf{K}}$ depends strongly on the bandwidth, which in turn depends on the point numbering (Duff and Meurant (1989), Venkatakrishnan and Mavriplis (1993, 1995)). The smaller the bandwidth, the closer $\tilde{\mathbf{K}}$ is to $\mathbf{K}$, and the better the preconditioning. This is to be expected for problems with no discernable stiffness direction. If, on the other hand, a predominant stiffness direction exists, the point numbering should be aligned with it. This may or may not result in small bandwidths (see Figure 7.3 for a counterexample), but is certainly the most advisable way to renumber the points.

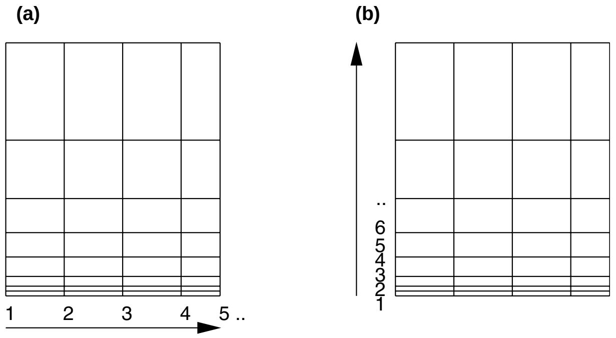


Figure 7.3. Counterexample

Before going on, the reader should consider the storage requirements of ILU preconditioners. Assuming the lower bound of no allowed fill-in (`nfilr=0`), a discretization of space using linear tetrahedra and `neqns` unknowns per point, we require `nstor=2*neqns*neqns*nedge` for $\tilde{\mathbf{L}}$, $\tilde{\mathbf{U}}$, which for the Euler or laminar Navier–Stokes equations with `neqns=5` and on a typical mesh with `nedge=7*npoin` translates into `nstor=350*npoin` storage locations. Given that a typical explicit Euler solver on the same type of grid only requires `nstor=90*npoin` storage locations, it is not difficult to see why even one more layer of fill-in is seldom used.

7.2.1.5. Block methods

Considering that the cost of direct solvers scales with the square of the bandwidth, another possibility is to decompose **K** into blocks. These blocks are then solved for directly. The reduction in cost is a result of neglecting all matrix entries outside the block, leading to lower bandwidths. With the notation of Figure 7.4, we may decompose **K** additively as

$$\mathbf{K} = \mathbf{L}^b + \mathbf{D}^b + \mathbf{U}^b \tag{7.31}$$

or as a product of block matrices:

$$\mathbf{K} = \prod_{j=1}^{m} \mathbf{K}_j . \tag{7.32}$$

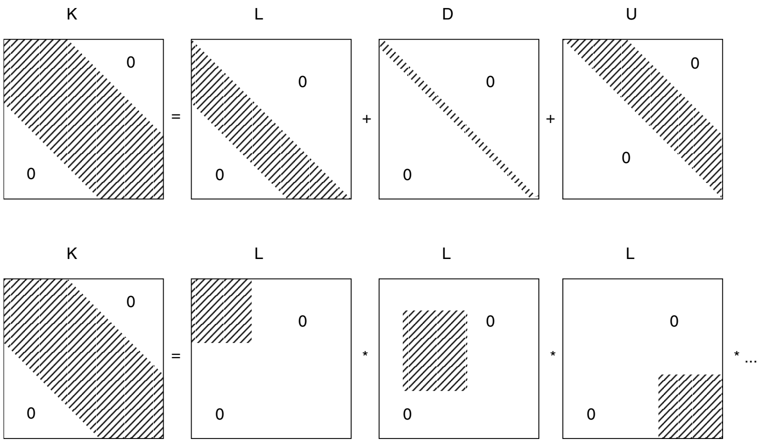


Figure 7.4. Decomposition of a matrix

For the additive decomposition, one can either operate without reusing the unknowns at the solution stage, i.e. just on the diagonal level,

$$\tilde{\mathbf{K}} = \mathbf{D}^b, \tag{7.33}$$

or, analogous to Gauss–Seidel, by reusing the unknowns at the solution stage,

$$\tilde{\mathbf{K}}^L = \mathbf{L}^b + \mathbf{D}^b, \quad \tilde{\mathbf{K}}^U = \mathbf{U}^b + \mathbf{D}^b. \tag{7.34}$$

For the product decomposition, the preconditioner is of the form

$$\tilde{\mathbf{K}} = \mathbf{D}^{\frac{1}{2}} \left[\prod_{j=1}^{m} (\mathbf{I} + \mathbf{E}_j^b) \right] \mathbf{D}^{\frac{1}{2}}, \tag{7.35}$$

where **I** denotes the identity matrix, and **E** contains the off-diagonal block entries, scaled by **D**. As before, the propagation of information is determined by the numbering of the blocks. Typical examples of this type of preconditioning are element-by-element (Hughes *et al.* (1983a,c)) or group-by-group (Tezduyar and Liou (1989), Tezduyar *et al.* (1992a), Liou and Tezduyar (1992)) techniques.

7.2.2. GLOBALIZATION PROCEDURES

As seen from the previous section, any form of preconditioning neglects some information from the original matrix $\mathbf{K}$. The result is that after an initially fast convergence, a very slow rate of convergence sets in. In order to avert this behaviour, a number of acceleration or globalization procedures have been devised. The description that follows starts with the analytical ones, and then proceeds to matrix-based and grid-based acceleration. Let us recall the basic iterative scheme:

$$\tilde{\mathbf{K}} \cdot (\mathbf{u}^{n+1} - \mathbf{u}^{n}) = \tilde{\mathbf{K}} \cdot \Delta\mathbf{u} = \Delta\tau \cdot \mathbf{r} = \Delta\tau \cdot (\mathbf{f} - \mathbf{K} \cdot \mathbf{u}). \tag{7.36}$$

7.2.2.1. Tchebichev acceleration

This type of acceleration procedure may best be explained by considering the matrix system that results from the discretization of the Laplace operator on a grid of linear elements of constant size h_x, h_y, h_z. If the usual 3/5/7-star approximation to the Laplacian

$$\nabla^2 u = 0 \tag{7.37}$$

is employed, the resulting discretization at node i, j, k for the Jacobi iterations with $\tilde{\mathbf{K}} = \mathbf{D}$ and $\Delta\tau = \Delta t$ takes the form

$$\begin{aligned} 4(1 + a^2 + b^2)\Delta\mathbf{u}_{i,j,k} = \Delta t[&(u_{i-1,j,k} - 2u_{i,j,k} + u_{i+1,j,k}) \\ &+ a^2(u_{i,j-1,k} - 2u_{i,j,k} + u_{i,j+1,k}) \\ &+ b^2(u_{i,j,k-1} - 2u_{i,j,k} + u_{i,j,k+1})], \end{aligned} \tag{7.38}$$

with $a = h_x/h_y$, $b = h_x/h_z$. Inserting the Fourier mode

$$u = g^p_{m,n,l} \exp\left(\frac{i\pi x}{mh_x}\right) \exp\left(\frac{j\pi y}{nh_y}\right) \exp\left(\frac{k\pi z}{lh_z}\right) \tag{7.39}$$

yields a decay factor $g_{m,n,l}$ between iterations of the form

$$g_{m,n,l} = 1 - \Delta t f(a, b, m, n, l), \tag{7.40}$$

with

$$\begin{aligned} f(a, b, m, n, l) = \frac{1}{2(1 + a^2 + b^2)}\bigg[&\left(1 + \cos\frac{\pi}{m}\right) + a^2\left(1 + \cos\frac{\pi}{n}\right) \\ &+ b^2\left(1 + \cos\frac{\pi}{l}\right)\bigg]. \end{aligned} \tag{7.41}$$

Note that we have lumped the constant portions of the grid and a mode into the function $f(a, b, m, n, l)$. After p timesteps with varying Δt, the decay factor will be given by

$$g_{m,n,l} = \prod_{q=1}^{p} [1 - \Delta t_q f(a, b, m, n, l)]. \tag{7.42}$$

The objective is to choose a sequence of timesteps so that as many eigenmodes as possible are reduced. The following two sequences have been used with success to date:

(a) *Tchebicheff sequence (Löhner and Morgan (1987))*:

$$\Delta t_q = \frac{2}{1 + \cos[\pi \cdot (q-1)/p]}, \quad q = 1, \ldots, p; \tag{7.43a}$$

(b) *superstep sequence (Gentzsch and Schlüter (1978), Gentzsch (1980))*:

$$\Delta t_q = \frac{2}{1 + (R/p^2) + \cos[\pi \cdot (2q-1)/2p]}, \quad q = 1, \ldots, p, \ R = 2.5. \tag{7.43b}$$

Observe that in both cases the maximum timestep is of order $\Delta t = O(p^2)$, which is outside the stability range. The overall procedure nevertheless remains stable, as the smaller timesteps 'rescue' the stability. Figure 7.5 compares the performances of the two formulas for the 1-D case with that of uniform timesteps. The improvement in residual reduction achieved by the use of non-uniform timesteps is clearly apparent.

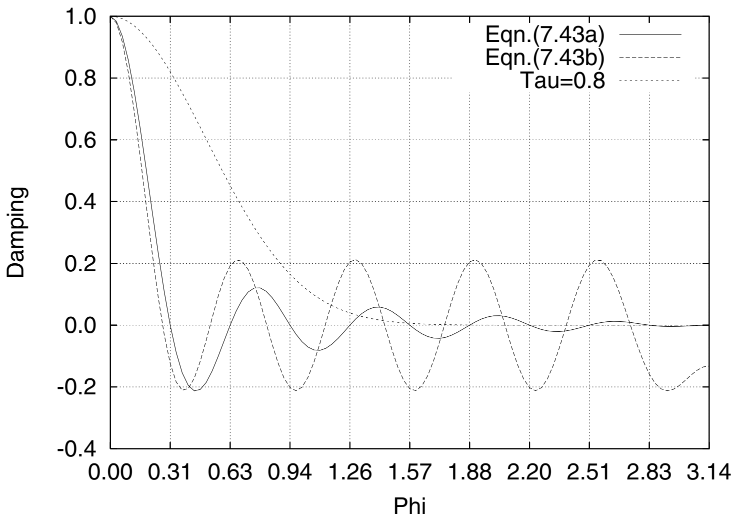


Figure 7.5. Damping curves for the Laplacian

Returning to (7.42), let us determine the magnitude of Δt required to eliminate a certain mode. For any given mode $g_{m,n,l}$, the mode can be eliminated by choosing a timestep of magnitude

$$\Delta t = \frac{2(1+c^2)}{(1+\cos(\pi/m)) + a^2(1+\cos(\pi/n)) + b^2(1+\cos(\pi/l))}, \tag{7.44}$$

with $c^2 = a^2 + b^2$. The timesteps required to eliminate the three highest modes have been summarized in Table 7.1. Two important trends may be discerned immediately.

(a) The magnitude of Δt, or, equivalently, the number of iterations required to eliminate the three highest modes, increases with the dimensionality of the problem. This is the case even for uniform grids ($a = b = 1$).

(b) For non-uniform grids, the magnitude of Δt necessary to eliminate modes increases quadratically with the aspect ratio of the elements. This is a reflection of the second-order derivatives that characterize the Laplace operator. For uniform timesteps/overrelaxation, the number of iterations would increase quadratically with the aspect ratio, whereas for varying Δt it increases linearly.

Table 7.1. Timestep required to eliminate mode $(\pi/n, \pi/m, \pi/l)$

`ndimn`	Modes	Δt	Modes	Δt	Modes	Δt
1	π	1	$\pi/2$	2	$\pi/3$	4
2	π, π	1	$\pi/2, \pi/2$	2	$\pi/3, \pi/3$	4
2	$\pi, 0$	$1+a^2$	$\pi/2, 0$	$2(1+a^2)$	$\pi/3, 0$	$4(1+a^2)$
3	π, π, π	1	$\pi/2, \pi/2, \pi/2$	2	$\pi/3, \pi/3, \pi/3$	4
3	$\pi, 0, 0$	$1+c^2$	$\pi/2, 0, 0$	$2(1+c^2)$	$\pi/3, 0, 0$	$4(1+c^2)$

The timestepping/overrelaxation sequence outlined above for the Laplace equation on uniform grids can also be applied to general, unstructured grids.

7.2.2.2. *Dominant eigenvalue acceleration*

Consider the preconditioned iterative scheme

$$\tilde{\mathbf{K}} \cdot (\mathbf{u}^{n+1} - \mathbf{u}^n) = \tilde{\mathbf{K}} \cdot \Delta\mathbf{u} = \Delta\tau \cdot \mathbf{r} = \Delta\tau \cdot (\mathbf{f} - \mathbf{K} \cdot \mathbf{u}). \tag{7.45}$$

This scheme may be interpreted as an explicit timestepping scheme of the form

$$\tilde{\mathbf{K}} \cdot \frac{d\mathbf{u}}{dt} + \mathbf{K} \cdot \mathbf{u} = \mathbf{f}, \tag{7.46}$$

or, setting $\mathbf{A} = \tilde{\mathbf{K}}^{-1}\mathbf{K}$,

$$\frac{d\mathbf{u}}{dt} + \mathbf{A} \cdot \mathbf{u} = \mathbf{g}. \tag{7.47}$$

Assuming $\mathbf{A}$, $\mathbf{g}$ constant, the solution to this system of ODEs is given by

$$\mathbf{u} = \mathbf{u}_\infty + e^{-\mathbf{A}t}(\mathbf{u}_0 - \mathbf{u}_\infty), \tag{7.48}$$

where $\mathbf{u}_0$, $\mathbf{u}_\infty$ denote the starting and steady-state values of $\mathbf{u}$, respectively. Assembling all eigenvalue equations

$$\mathbf{A} \cdot \mathbf{y}_i = \mathbf{y}_i \lambda_i \tag{7.49}$$

into one large matrix equation results in

$$\mathbf{A} \cdot \mathbf{Y} = \mathbf{Y} \cdot \mathbf{\Lambda}, \ \mathbf{\Lambda} = \begin{bmatrix} \lambda_{\min} & & & \\ & \cdot & & \\ & & \cdot & \\ & & & \lambda_{\max} \end{bmatrix}, \quad \mathbf{Y} = [\mathbf{y}_1, \mathbf{y}_2, \ldots] \tag{7.50}$$

or

$$\mathbf{A} = \mathbf{Y} \cdot \mathbf{\Lambda} \cdot \mathbf{Y}^{-1}. \tag{7.51}$$

We can now expand $e^{-\mathbf{A}t}$ as a series,

$$e^{-\mathbf{A}t} = \sum_{n=0}^{\infty} \frac{(-\mathbf{A}t)^n}{n!}. \tag{7.52}$$

The close inspection of a typical term in this series reveals that

$$\frac{(-\mathbf{A}t)^n}{n!} = \frac{(-t)^n}{n!}[\mathbf{Y} \cdot \mathbf{\Lambda} \cdot \mathbf{Y}^{-1}]^n = \mathbf{Y}\frac{(-\mathbf{\Lambda}t)^n}{n!}\mathbf{Y}^{-1}, \tag{7.53}$$

implying that

$$e^{-\mathbf{A}t} = \mathbf{Y}e^{-\mathbf{\Lambda}t}\mathbf{Y}^{-1}. \tag{7.54}$$

We can therefore rewrite (7.48) as

$$\mathbf{u} = \mathbf{u}_\infty + \mathbf{Y}e^{-\mathbf{\Lambda}t}\mathbf{Y}^{-1}(\mathbf{u}_0 - \mathbf{u}_\infty). \tag{7.55}$$

As time or equivalently the number of iterations in (7.45) increases, the lowest eigenvalues will begin to dominate the convergence to a steady state or equivalently the solution. As the solution reaches a steady state ($t \to \infty$), the solution approaches

$$\mathbf{u} - \mathbf{u}_\infty = \mathbf{a}e^{-\lambda_{\min} t}, \quad \mathbf{a} = \mathbf{u}_0 - \mathbf{u}_\infty. \tag{7.56}$$

If a way can be found to somehow detect this eigenvalue and its associated eigenvector, an acceleration procedure may be devised. By differentiating (7.56) we obtain

$$\Delta\mathbf{u} = -\lambda_{\min}\Delta t e^{-\lambda_{\min} t}\mathbf{a} = -\lambda_{\min}\Delta t(\mathbf{u} - \mathbf{u}_\infty), \tag{7.57}$$

implying that

$$\mathbf{u}_\infty = \mathbf{u} + \frac{1}{\lambda_{\min}\Delta t}\Delta\mathbf{u} = \mathbf{u} + \alpha\Delta\mathbf{u}, \tag{7.58}$$

with

$$\alpha = \frac{1}{\lambda_{\min}\Delta t}. \tag{7.59}$$

Given that α is a scalar, and we are trying to infer information from a vector, many possibilities exist to determine it. The most natural choice is to differentiate (7.57) once more to obtain the second differences

$$\Delta(\Delta\mathbf{u}) = -\lambda_{\min}\Delta t\Delta\mathbf{u}, \tag{7.60}$$

and then dot (i.e. weigh) this equation with either $\Delta\mathbf{u}$ or $\Delta(\Delta\mathbf{u})$ to obtain

$$\alpha = \frac{1}{\lambda_{\min}\Delta t} = -\frac{\Delta\mathbf{u} \cdot \Delta\mathbf{u}}{\Delta\mathbf{u} \cdot \Delta(\Delta\mathbf{u})}, \tag{7.61a}$$

$$\alpha = \frac{1}{\lambda_{\min}\Delta t} = -\frac{\Delta(\Delta\mathbf{u}) \cdot \Delta\mathbf{u}}{\Delta(\Delta\mathbf{u}) \cdot \Delta(\Delta\mathbf{u})}. \tag{7.61b}$$

Either of these two procedures can be used. The application of an overrelaxation factor α to augment the basic iterative procedure is only useful once the low-frequency modes start to dominate. This implies that the value of α should not change appreciably between iterations.

Typically, this dominant-eigenvalue acceleration is applied if two consecutive values of α do not differ by more than a small amount, i.e.

$$\left|\frac{\Delta\alpha}{\alpha}\right| < c, \tag{7.62}$$

where, typically, $c \leq 0.05$ (Zienkiewicz and Löhner (1985)). The effect of applying this simple procedure for positive definite matrix systems stemming from second-order operators like the Laplacian is a reduction of the asymptotic number of iterations from $O(N_g^2)$ to $O(N_g)$, where N_g is the graph depth associated with the spatial discretization of the problem. An extension of this technique to non-symmetric matrix systems was proposed by Hafez *et al.* (1985).

7.2.2.3. Conjugate gradients

For symmetric positive systems, conjugate gradient algorithms (Hestenes and Stiefel (1952)) are commonly used, as they offer the advantages of easy programming, low memory overhead, and excellent vectorization and parallelization properties. Rewriting the original system of equations

$$\mathbf{K}\cdot\mathbf{u} = \mathbf{f} \tag{7.63}$$

leads to the basic iterative step given by

$$\Delta\mathbf{u}^k = \alpha_k(-\mathbf{r}^{k-1} + e_{k-1}\Delta\mathbf{u}^{k-1}) = \alpha_k\mathbf{v}^k, \tag{7.64}$$

where $\Delta\mathbf{u}^k$, $\mathbf{r}^{k-1}$ denote the increment and residual ($\mathbf{r} = \mathbf{f} - \mathbf{K}\cdot\mathbf{u}$) vectors respectively, and α_k, e_{k-1} are scaling factors. Observe that the biggest difference between this procedure and the simple iterative scheme given by (7.45) is the appearance of a second search direction and a corresponding scaling factor e_{k-1}. The choice of e_{k-1} is performed in such a way that the increments of two successive steps are orthogonal with respect to some norm. The natural norm for (7.64) is the matrix-norm $\mathbf{K}$, i.e.

$$\Delta\mathbf{u}^{k-1}\cdot\mathbf{K}\cdot\Delta\mathbf{u}^k = 0. \tag{7.65}$$

This immediately leads to

$$e_{k-1} = -\frac{\mathbf{r}^{k-1}\cdot\mathbf{K}\cdot\Delta\mathbf{u}^{k-1}}{\Delta\mathbf{u}^{k-1}\cdot\mathbf{K}\cdot\Delta\mathbf{u}^{k-1}}. \tag{7.66}$$

This expression may be simplified further by observing that

$$\mathbf{r}^{k-1} - \mathbf{r}^{k-2} = -\mathbf{K}\cdot\Delta\mathbf{u}^{k-1}, \tag{7.67}$$

yielding

$$e_{k-1} = \frac{\mathbf{r}^{k-1}\cdot(\mathbf{r}^{k-1} - \mathbf{r}^{k-2})}{\Delta\mathbf{u}^{k-1}\cdot(\mathbf{r}^{k-1} - \mathbf{r}^{k-2})}. \tag{7.68}$$

The scaling factor α_k is obtained similarly to (7.66), i.e. by forcing

$$\mathbf{K}\cdot(\mathbf{u}^{k-1} + \Delta\mathbf{u}^k) = \mathbf{f} \tag{7.69}$$

in a ‘matrix weighted’ sense by multiplication with $\Delta\mathbf{u}^k$:

$$\Delta\mathbf{u}^k \cdot \mathbf{K} \cdot \Delta\mathbf{u}^k = \Delta\mathbf{u}^k \cdot \mathbf{r}^{k-1}. \tag{7.70}$$

Upon insertion of (7.64) into (7.70) we obtain

$$\alpha_k = \frac{\mathbf{v}^k \cdot \mathbf{r}^{k-1}}{\mathbf{v}^k \cdot \mathbf{K} \cdot \mathbf{v}^k}. \tag{7.71}$$

The amount of work required during each iteration consists of several scalar products and one matrix–vector multiplication which is equivalent to a RHS evaluation. Theoretically, the conjugate gradient algorithm will converge to the exact solution in at most N_{eq} iterations, where N_{eq} is the number of equations to be solved. However, conjugate gradient algorithms are only of interest because they usually converge much faster than this pessimistic estimate. On the other hand, the transfer of information between the unknowns only occurs on the RHS, and therefore, for a problem with graph depth N_g, the minimum algorithmic complexity of the conjugate gradient algorithm is of $O(N_g \cdot N_{eq})$. The only way to reduce this complexity is through preconditioning procedures that go beyond nearest-neighbour information.

7.2.2.4. Generalized minimal residuals

For unsymmetric matrices, such as those that arise commonly when discretizing the Euler and Navier–Stokes equations, the conjugate gradient algorithm will fail to produce acceptable results. The main reason for this failure is that, with only two free parameters α_k, e_k, no complex eigenvectors can be treated properly. In order to be able to treat such problems, the space in which the new increment $\Delta\mathbf{u}^k$ is sought has to be widened from the two vectors $\mathbf{r}^{k-1}$, $\Delta\mathbf{u}^{k-1}$. This vector subspace is called a Krylov space. We will denote the vectors spanning it by $\mathbf{v}^k$, $k = 1, m$. In order to construct an orthonormal basis in this space, the following Gram–Schmidt procedure is employed:

(a) starting vector (= residual)

$$\mathbf{v}^1 = \frac{\mathbf{r}^n}{|\mathbf{r}^n|}, \quad \mathbf{r}^n = \mathbf{f} - \mathbf{K} \cdot \mathbf{u}^n; \tag{7.72}$$

(b) for $j = 1, 2, \ldots, m-1$ take

$$\mathbf{w}^{j+1} = \mathbf{K} \cdot \mathbf{v}^j - \sum_{i=1}^{j} h^{ij}\mathbf{v}^i, \quad h^{ij} = \mathbf{v}^i \cdot \mathbf{K} \cdot \mathbf{v}^j; \tag{7.73}$$

$$\mathbf{v}^{j+1} = \frac{\mathbf{w}^{j+1}}{|\mathbf{w}^{j+1}|}. \tag{7.74}$$

Observe that, like all other iterative procedures, the first vector lies in the direction of the residual. Moreover,

$$\mathbf{v}^l \cdot \mathbf{v}^{j+1} = \frac{1}{|\mathbf{w}^{j+1}|}\left[\mathbf{v}^l \cdot \mathbf{K} \cdot \mathbf{v}^j - \sum_{i=1}^{j} (\mathbf{v}^i \cdot \mathbf{K} \cdot \mathbf{v}^j)\mathbf{v}^l \cdot \mathbf{v}^i\right]. \tag{7.75}$$

Assuming that the first j vectors are orthonormal, then the only inner product left for the last expression is the one for which $l = i$, whence

$$\mathbf{v}^l \cdot \mathbf{v}^{j+1} = \frac{1}{|\mathbf{w}^{j+1}|}[\mathbf{v}^l \cdot \mathbf{K} \cdot \mathbf{v}^j - \mathbf{v}^l \cdot \mathbf{K} \cdot \mathbf{v}^j] = 0, \tag{7.76}$$

which proves the orthogonalization procedure. When the set of search vectors $\mathbf{v}^k$, $k = 1, m$ has been constructed, the increment in the solution is sought as a linear combination

$$\Delta\mathbf{u} = \mathbf{v}^k a_k, \tag{7.77}$$

in such a way that the residual is minimized in a least-squares sense (hence the name generalized minimal residuals (Saad and Schultz (1986), Wigton *et al.* (1985), Venkatakrishnan (1988), Saad (1989), Venkatakrishnan and Mavriplis (1993))

$$|\mathbf{K} \cdot (\mathbf{u} + \Delta\mathbf{u}) - \mathbf{f}|^2 \rightarrow \min, \tag{7.78}$$

or

$$|\mathbf{K} \cdot (\mathbf{v}^k a_k) - \mathbf{r}^n|^2 \rightarrow \min. \tag{7.79}$$

The solution to this minimization problem leads to the matrix problem

$$A^{kl} a_l = (\mathbf{K} \cdot \mathbf{v}^k) \cdot (\mathbf{K} \cdot \mathbf{v}^l) a_l = (\mathbf{K} \cdot \mathbf{v}^k) \cdot \mathbf{r}^n = b^k, \tag{7.80}$$

or

$$\mathbf{A} \cdot \mathbf{a} = \mathbf{b}. \tag{7.81}$$

The amount of work required for each GMRES iteration with m search directions consists of $O(m^2)$ scalar products and $O(m)$ matrix–vector multiplications which are equivalent to a RHS evaluation. As with the conjugate gradient method, the transfer of information between the unknowns only occurs on the RHS, and therefore, for a model problem with graph depth N_g, the minimum algorithmic complexity of the GMRES algorithm is of $O(N_g \cdot N_{eq})$. The only way to reduce this complexity is through preconditioning procedures that go beyond nearest-neighbour information (e.g. using LU-SGS, see Luo *et al.* (1998)).

7.3. Multigrid methods

Among the various ways devised to solve efficiently a large system of equations of the form

$$\mathbf{K} \cdot \mathbf{u} = \mathbf{f}, \tag{7.82}$$

multigrid solvers are among the most efficient. Their theoretical algorithmic complexity is only of $O(N_{eq} \log N_{eq})$, which is far better than direct solvers, GMRES or conjugate gradient iterative solvers, or any other solver for that matter. The concept of the algorithm, which is grid-based but may be extended to a more general matrix-based technique called algebraic multigrid (AMG), dates back to the 1960s (Fedorenko (1962, 1964)). The first successful application of multigrid techniques in CFD was for the solution of potential flow problems given by the Laplace or full potential equations (Jameson and Caughey (1977), Jameson (1979)). The concept was extended a decade later to the Euler equations

(Jameson *et al.* (1981), Jameson and Yoon (1985), Mavriplis and Jameson (1987), Mavriplis (1991b)) and developed further for the RANS equations (Rhie (1986), Martinelli and Jameson (1988). Alonso *et al.* (1995), Mavriplis (1995, 1996), Mavriplis *et al.* (2005)). The present section is intended as an introduction to the concept. For a thorough discussion, see Hackbusch and Trottenberg (1982), Brand (1983), Ruge and Stüben (1985), Trottenberg *et al.* (2001) and Wesseling (2004). Some example cases that show the versatility of the technique are also included.

7.3.1. THE MULTIGRID CONCEPT

The basic concept of any multigrid solver may be summarized as follows. Given a system of equations like (7.82) to be solved on a fine grid denoted by the superscript *fg*

$$\mathbf{K}^{fg} \cdot \mathbf{u}^{fg} = \mathbf{f}^{fg}, \tag{7.83}$$

solve iteratively for $\mathbf{u}^{fg}$ until the residual

$$\mathbf{r}^{fg} = \mathbf{f}^{fg} - \mathbf{K}^{fg} \cdot \mathbf{u}^{fg} \tag{7.84}$$

is smooth. This is usually achieved after a few (<10) iterations. At this stage, any further reduction of $\mathbf{r}^{fg}$ becomes more and more costly as the lower-order eigenmodes of $\mathbf{r}^{fg}$ involve more than nearest-neighbour connections in $\mathbf{K}^{fg}$. In order to avoid this increase in cost, the residual is transferred to a coarser grid via a so-called injection operator Q:

$$\mathbf{K}^{cg} \cdot \mathbf{v}^{cg} = Q^{cg}_{fg} \cdot \mathbf{r}^{fg} = \mathbf{f}^{cg}. \tag{7.85}$$

This equation may be restated in the more familiar form given by (7.82) by transferring as well the current fine-grid values of the unknowns to the coarser grid:

$$\mathbf{K}^{cg} \cdot (\mathbf{u}^{cg}_0 + \mathbf{v}^{cg}) = \mathbf{f}^{cg} + \mathbf{K}^{cg} \cdot \mathbf{u}^{cg}_0, \quad \mathbf{u}^{cg}_0 = Q^{cg}_{fg} \cdot \mathbf{u}^{fg}. \tag{7.86}$$

This last equation may now be restated as

$$\mathbf{K}^{cg} \cdot \mathbf{u}^{cg} = \mathbf{r}^{cg}, \tag{7.87}$$

which is of the same form as (7.83). If we assume that on this coarse grid the solution of $\mathbf{u}^{cg}$ has been obtained, the increment obtained is added back to the fine-grid solution via a so-called projection operator P:

$$\mathbf{u}^{fg} \leftarrow \mathbf{u}^{fg} + P^{fg}_{cg} \cdot (\mathbf{u}^{cg} - \mathbf{u}^{cg}_0).$$

This two-grid procedure can be generalized to a whole sequence of multiple grids, hence the name multigrid. For each of the coarser grids, the problem given by (7.87) is treated in the same manner as (7.83), i.e. the coarse grid becomes the fine grid of even coarser grids, and so on.

At this stage, the very general concept of multigrid has been defined. We now proceed to detail the several aspects that comprise a multigrid solver: the injection and projection operators, how the different grids are visited during the solution, pre- and post-smoothing options, and the algorithmic complexity of the overall procedure.

7.3.2. INJECTION AND PROJECTION OPERATORS

Multigrid solvers, by their very nature, require the transfer of information between different grids. In the most general terms, given two grids with unknowns $\mathbf{u}^1$, $\mathbf{u}^2$, where $\mathbf{u}^1$ is known, the transfer of information is accomplished by setting

$$u^2 = N_2^i \hat{u}_i^2 \approx N_1^j \hat{u}_j^1 = u^1, \tag{7.88}$$

where N_1^j, N_2^i denote the shape functions of grids 1, 2, and $\hat{u}_j^1$, $\hat{u}_i^2$ the respective components of $\mathbf{u}^1$, $\mathbf{u}^2$. This approximation may now be treated in the same way as discussed in Chapter 4 for approximation theory. The most natural choice, which minimizes the L2-norm, is to take as weighting functions the shape functions of the second grid:

$$\int N_2^k N_2^i \, d\Omega \, \hat{u}_i^2 = \int N_2^k N_1^j \, d\Omega \, \hat{u}_j^1. \tag{7.89}$$

On the LHS the familiar consistent mass matrix $\mathbf{M}_{c2}$ is obtained. On the RHS, a new integral appears that consists of the inner product of shape functions from different grids. This integral is fairly complicated, particularly for sets of non-nested tetrahedral or hexahedral grids. For nested grids in one dimension, we obtain the following injection and projection operators (see Figure 7.6):

(a) projection (nested 1-D):

$$\frac{2h}{6}(u_{i-1}^2 + 4u_i^2 + u_{i+1}^2) = \frac{h}{24}(u_{j-2}^1 + 6u_{j-1}^1 + 10u_j^1 + 6u_{j+1}^1 + u_{j+2}^1); \tag{7.90}$$

(b) injection (nested 1-D):

$$\frac{h}{6}(u_{i-1}^1 + 4u_i^1 + u_{i+1}^1) = \frac{2h}{24}(6u_{j-1}^2 + 6u_j^2), \tag{7.91}$$

$$\frac{h}{6}(u_{i-1}^1 + 4u_i^1 + u_{i+1}^1) = \frac{2h}{24}(u_{j-1}^2 + 10u_j^2 + u_{j+1}^2). \tag{7.92}$$

For typical multigrid sequences, where the assumption of strict coarsening can be made, the integral appearing on the RHS of (7.89) is usually avoided. Instead, straight interpolation and projection are employed.

The injection operation, i.e. going from a finer to a coarser grid, is accomplished by locating the host element of the coarse grid for each of the points of the fine grids, and sending the appropriately weighted portions of the unknowns to each of the nodes of the host element. This procedure has been sketched in Figure 7.7(a). The unknowns are weighted by their respective areas. This implies that (7.89) is approximated by

$$\mathbf{M}_{cg}\mathbf{u}^{cg} = \sum_{pfg} W_{pcg}^{pfg} V_{pfg} \hat{u}_{pfg}. \tag{7.93}$$

Here W_{pcg}^{pfg} denotes the weighting factor for any of the points of the fine grid with respect to the points of the host element of the coarse grid, and V_{pfg} is the volume associated with any of the points of the fine grid (equivalent to the lumped mass matrix). Given that the weights

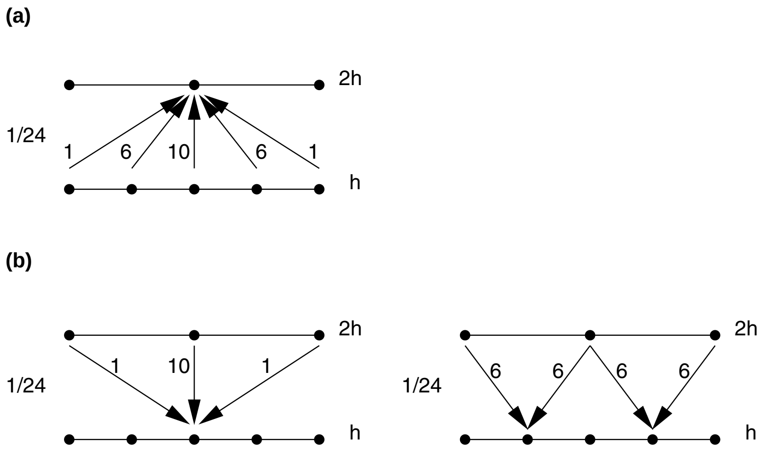


Figure 7.6. (a) Projection and (b) injection operators for 1-D grids

W_{pcg}^{pfg} of the nodes for any of the host elements add up to unity, this transfer of information is conservative.

The projection operation, i.e. going from a coarser to a finer grid, is simplified to a straight interpolation. For each of the points of the fine mesh, the host element of the coarse grid is identified and the solution is interpolated. This procedure, which may be written as

$$\mathbf{u}^{fg} = \sum_{pcg} W_{pcg}^{pfg} \hat{u}_{pcg}, \tag{7.94}$$

has been sketched in Figure 7.7(b). Observe that this is a considerable simplification over (7.93).

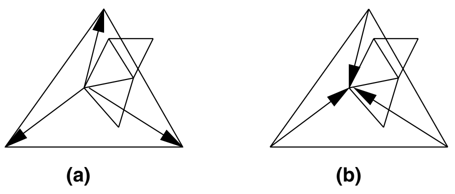


Figure 7.7. (a) Projection (fine to coarse) and (b) injection (coarse to fine) operators

The reader should also realize that, for a general transfer of information between grids, neither of the two simplifications just described is useful. If we consider the simple 1-D grid transfer problem shown in Figure 7.8, one may be tempted to use the simplified projection operator for the left half of the domain, but this would certainly fail for the right half. Conversely, interpolation would fail for the left half of the domain. This case clearly illustrates the need to employ more elaborate projection and injection procedures for general grid transfer operations.

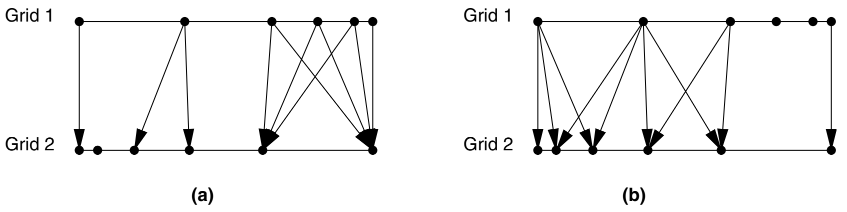


Figure 7.8. (a) Projection and (b) injection operators for 1-D grids

7.3.3. GRID CYCLING

The general multigrid procedure described above left many possibilities for the way in which grids are visited during the solution. The most basic of these cycles is the so-called V-cycle, illustrated in Figure 7.9(a). Starting from the finest grid, all subsequent coarser grids are visited only once. The solution is then projected to the finer grids, completing the cycle. Alternatively, one could do more work on the coarser grids in an attempt to drive the residuals down as much as possible before returning to the expensive finer grids. This would then lead to the so-called W-cycle, illustrated in Figure 7.9(b).

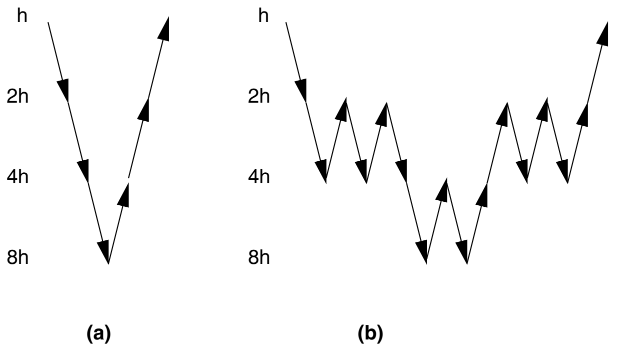


Figure 7.9. (a) V- and (b) W-cycles

The initial solution on the fine grid may be far from the final one, leading to unnecessary work. In order to obtain a better starting solution, the initial values of the unknowns are obtained on the next coarser grid. Repeating this procedure recursively leads to the so-called full multigrid cycle, sketched in Figure 7.10.

7.3.4. ALGORITHMIC COMPLEXITY AND STORAGE REQUIREMENTS

Let us analyse the algorithmic complexity and the storage requirements of multigrid solvers. For a problem in d dimensions, the reduction in the number of gridpoints between subsequent grids will be of the order of $N_{cg}/N_{fg} = 1/2^d$. If we assume that the same number of iterative smoothing passes n_{it} are required on each grid and that the work in each of these passes is proportional to the number of points, then the work required for a V-cycle is of order

$$W_V \approx n_{it} N_{fg} \log_d(N_{fg}), \tag{7.95}$$

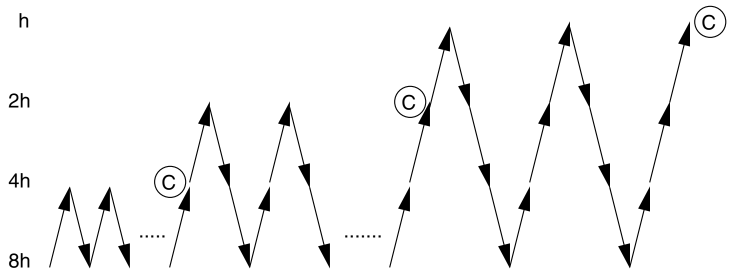


Figure 7.10. Full multigrid procedure

with associated storage requirements of

$$S = N_{fg} \log_d(N_{fg}). \tag{7.96}$$

One can see that these work and storage estimates are extremely favourable. In fact, it is difficult to imagine that to solve (7.82) any smaller work estimates can be obtained, regardless of the method. For the W-cycle, the amount of work is only slightly larger, as the number of gridpoints on the coarser grids drops dramatically, particularly in three dimensions.

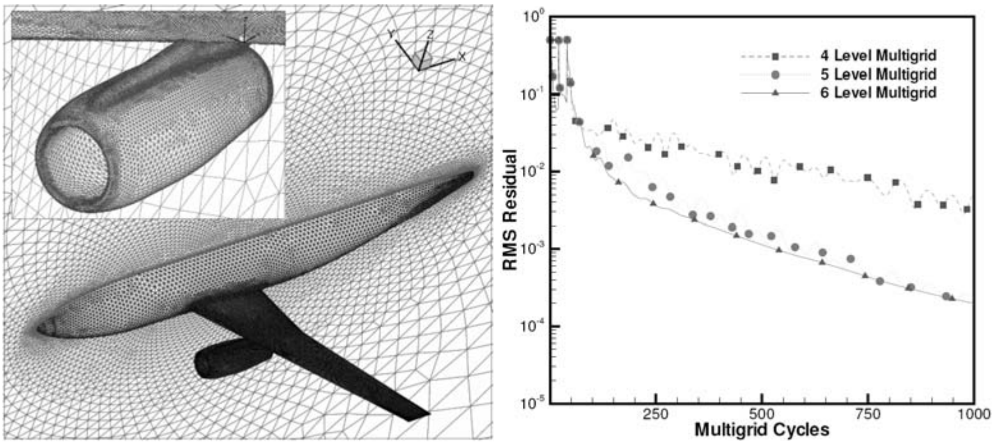


Figure 7.11. Performance of a multigrid for external aerodynamics

7.3.5. SMOOTHING

The whole concept of multigrid methods hinges on the notion that efficient smoothers are available. The smoother employed must be able to reduce the high-frequency content of the residual significantly in only a few iterative steps. The smoothers most commonly employed are as follows.

(a) *Point Jacobi*. This is an obvious choice for *elliptic operators* on non-stretched grids. When augmented with the timestepping/overrelaxation sequences given by (7.43a,b) it leads to very efficient smoothers.

(b) *Runge–Kutta*. This has been a popular way of smoothing the residuals of *hyperbolic operators* on uniform and stretched grids (Jameson (1979), Mavriplis and Jameson (1987), Martinelli and Jameson (1988), Mavriplis (1991b, 1995, 1996)).

(c) *Incomplete LU*. This is a good smoother for problems that are of mixed type, and will work for both elliptic and hyperbolic operators (Jameson and Yoon (1985)).

The first two strategies are point-based, implying they do not go beyond nearest-neighbour connections for each pass. Therefore, they do not work efficiently for problems with highly stretched grids, diffusion tensors that show a significant degree of anisotropy and other 'hard' problems. For these classes of problems it is often advantageous to use line, plane, snake, or linelet relaxation procedures to smooth the residuals more efficiently (Hassan *et al*. (1990), Martin and Löhner (1992), Mavriplis (1996), Pierce *et al*. (1997)).

7.3.6. AN EXAMPLE

A typical example of the performance achievable with advanced multigrid solvers is shown in Figure 7.11, taken from Mavriplis *et al*. (2005). The case had 315 Mtets (70 million points) and was solved using four, five and six levels of multigrid with W-cycle. For the boundary layer regions linelet preconditioning was used within the basic explicit Runke-Kutta smoother. The freestream Mach number was $M_\infty = 0.75$, the angle of attack $\alpha = 0°$, and the Reynolds number based on the mean aerodynamic chord was of $O(3 \times 10^6)$. For the five and six multigrid level runs, the solution was adequately converged in approximately 800 multigrid cycles, while the four-level multigrid run suffers from slower convergence. Note that the single grid case (i.e. fine grid only without multigrid) would be very slow to converge, requiring several hundred thousand (!) iterations for a mesh of this size.

8 SIMPLE EULER/NAVIER–STOKES SOLVERS

This chapter describes some simple numerical solution schemes for the compressible Euler/Navier–Stokes equations. The discussion is restricted to central difference and Lax–Wendroff schemes, with suitable artificial viscosities. The more advanced schemes that employ limiters, such as flux-corrected transport (FCT) or total variation diminishing (TVD) techniques, are discussed in subsequent chapters.

Let us recall the compressible Navier–Stokes equations:

$$\mathbf{u}_{,t} + \nabla \cdot (\mathbf{F}^a - \mathbf{F}^v) = 0, \tag{8.1}$$

where

$$\mathbf{u} = \begin{Bmatrix} \rho \\ \rho v_i \\ \rho e \end{Bmatrix}, \quad \mathbf{F}^a_j = \begin{Bmatrix} \rho v_j \\ \rho v_i v_j + p\delta_{ij} \\ v_j(\rho e + p) \end{Bmatrix}, \quad \mathbf{F}^v_j = \begin{Bmatrix} 0 \\ \sigma_{ij} \\ v_l \sigma_{lj} + kT_{,j} \end{Bmatrix}. \tag{8.2}$$

Here ρ, p, e, T, k and v_i denote the density, pressure, specific total energy, temperature, conductivity and fluid velocity in direction x_i, respectively. This set of equations is closed by providing an equation of state, e.g. for a polytropic gas

$$p = (\gamma - 1)\rho[e - \tfrac{1}{2} v_j v_j], \quad T = c_v[e - \tfrac{1}{2} v_j v_j], \tag{8.3a, b}$$

where γ and c_v are the ratio of specific heats and the specific heat at constant volume, respectively. Furthermore, the relationship between the stress tensor σ_{ij} and the deformation rate must be supplied. For water and almost all gases, Newton's hypothesis

$$\sigma_{ij} = \mu\left(\frac{\partial v_i}{\partial x_j} + \frac{\partial v_j}{\partial x_i}\right) + \lambda \frac{\partial v_k}{\partial x_k}\delta_{ij} \tag{8.4}$$

complemented with Stokes' hypothesis

$$\lambda = -\frac{2\mu}{3} \tag{8.5}$$

is an excellent approximation. The compressible Euler equations are obtained by neglecting the viscous fluxes, i.e. setting $\mathbf{F}^v = 0$. Schemes suitable for the Euler equations are described first. Thereafter, the discretization of the viscous fluxes is discussed.

Applied Computational Fluid Dynamics Techniques: An Introduction Based on Finite Element Methods, Second Edition.
Rainald Löhner © 2008 John Wiley & Sons, Ltd. ISBN: 978-0-470-51907-3

8.1. Galerkin approximation

We start with the simplest possible scheme, i.e. straightforward Galerkin. Thus, we weight (8.1) with the available set of shape functions N^i:

$$\int_\Omega N^i(\mathbf{u}^h_{,t} + \nabla \cdot \mathbf{F})\, d\Omega = 0 \Rightarrow \int_\Omega N^i[N^j(\hat{\mathbf{u}}_j)_{,t} + \nabla \cdot \mathbf{F}(N^j\hat{\mathbf{u}}_j)]\, d\Omega = 0. \tag{8.6}$$

In order to minimize the algebra (and CPU) involved, one may use, without noticeable deterioration of results,

$$\mathbf{F}(N^j\hat{\mathbf{u}}_j) = N^j\mathbf{F}(\hat{\mathbf{u}}_j), \tag{8.7}$$

which then translates into

$$\int_\Omega N^i N^j d\Omega (\hat{\mathbf{u}}_j)_{,t} + \int_\Omega N^i \nabla \cdot N^j \, d\Omega\, \mathbf{F}(\hat{\mathbf{u}}_j) = 0, \tag{8.8}$$

or

$$\mathbf{M}_c \cdot \hat{\mathbf{u}}_{,t} = \mathbf{r}, \quad \mathbf{r} = \mathbf{r}(\mathbf{u}). \tag{8.9}$$

Obviously, integration by parts is possible for (8.8). For linear elements, one can show that this is equivalent to a FVM (see below). All integrals are again evaluated using the ***element sub-domain paradigm***

$$\int_\Omega \cdots = \sum_{el} \int_{\Omega_{el}} \cdots . \tag{8.10}$$

To obtain more insight into the resulting scheme, let us consider the linear advection equation in one dimension,

$$u_{,t} + au_{,x} = 0, \tag{8.11}$$

on a mesh with linear elements of uniform length h. For each element we have

$$\mathbf{M}_c = \int N^i N^j \, dx = \frac{h}{6}\begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}, \tag{8.12a}$$

$$\mathbf{D}_1 = a\int N^i N^j_{,x} = \frac{a}{2}\begin{bmatrix} -1 & 1 \\ -1 & 1 \end{bmatrix}, \tag{8.12b}$$

$$\mathbf{D}_2 = -\int N^i_{,x} N^j_{,x} \, dx = -\frac{1}{h}\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}. \tag{8.12c}$$

Assembly of all element contributions yields

$$\frac{h}{6}\begin{bmatrix} 2 & 1 & & \\ 1 & 4 & 1 & \\ & 1 & 4 & 1 \\ & & 1 & 2 \end{bmatrix} \cdot \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix}_{,t} = \frac{-a}{2}\begin{bmatrix} -1 & 1 & & \\ -1 & 0 & 1 & \\ & -1 & 0 & 1 \\ & & -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix}. \tag{8.13}$$

If mass lumping is performed, one obtains for node i

$$(u_i)_{,t} = -\frac{a}{2h}(u_{i+1} - u_{i-1}). \tag{8.14}$$

This is the same as central differencing! A stability analysis of the RHS,

$$r_i = -\frac{1}{2h}(u_{i+1} - u_{i-1}), \tag{8.15}$$

indicates that only every second node is coupled, allowing zero-energy or chequerboard modes in the solution. Therefore, stabilizing terms have to be added to re-couple neighbouring nodes.

The two most common stabilizing terms added are as follows.

1. *Terms of second order*: $h^2\, \partial^2/\partial x^2$, which result in an additional RHS term of

$$u_{i+1} - 2u_i + u_{i-1}, \tag{8.16}$$

which equals 4 for the $(-1, 1, -1)$ chequerboard pattern on a uniform mesh shown in Figure 8.1. Most TVD schemes use this type of stabilization term.

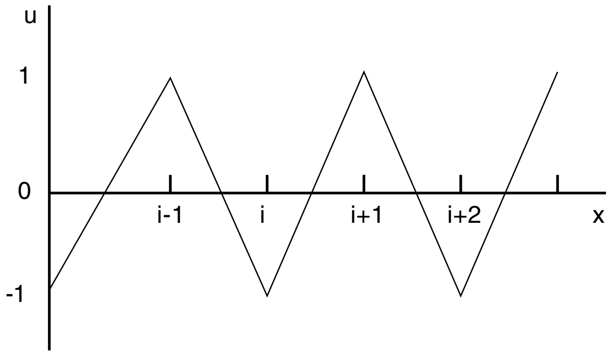


Figure 8.1. Chequerboard mode on a uniform 1-D grid

2. *Terms of fourth order*: $h^4\, \partial^4/\partial x^4$, which result in an additional RHS term of

$$u_{i+2} - 4u_{i+1} + 6u_i - 4u_{i-1} + u_{i-2}, \tag{8.17}$$

which equals 16 for the $(-1, 1, -1)$ chequerboard pattern on a uniform mesh shown in Figure 8.1. Observe that this type of stabilization term has a much more pronounced effect than second-order terms. Therefore, one may use much smaller constants when adding them. The fourth-order operator can be obtained in several ways. One obvious choice is to perform two ∇^2-passes over the mesh (Jameson *et al.* (1986), Mavriplis and Jameson (1990)). Another option is to first obtain the gradients of u at points, and then to approximate the third derivatives by taking a difference between first derivatives obtained from the gradients and the first derivatives obtained directly from the unknowns (Peraire *et al.* (1992a)). The implications of choosing either of these approaches will be discussed in more depth in Chapter 10.

8.1.1. EQUIVALENCY WITH FVM

Before going on, the GFEM using linear elements will be shown to be equivalent to the FVM. Integration by parts in (8.8) yields

$$\mathbf{r}^i = -\int_\Omega N^i \nabla \cdot N^j \mathbf{F}(\hat{\mathbf{u}}_j)\, d\Omega = \int_\Omega (\nabla N^i) N^j \cdot \mathbf{F}(\hat{\mathbf{u}}_j)\, d\Omega + \int_\Gamma \ldots . \tag{8.18}$$

As seen from Section 4.2.4, for *linear triangles*, this integral reduces to

$$\mathbf{r}^i = \frac{A}{3}(\nabla N^i) \cdot \sum_{j_{el}} \mathbf{F}(\hat{\mathbf{u}}_j) = \frac{1}{3} \cdot \frac{1}{2}(s_i \mathbf{n}_i) \cdot \sum_{j_{el}} \mathbf{F}(\hat{\mathbf{u}}_j), \tag{8.19}$$

and in particular, for node a in Figure 8.2,

$$\mathbf{r}_a = \frac{s_a \mathbf{n}_a}{3} \cdot \left(\frac{\mathbf{F}_a}{2} + \frac{\mathbf{F}_b + \mathbf{F}_c}{2} \right). \tag{8.20}$$

On the other hand,

$$\sum_{\mathrm{surr}_{el}} s_a \mathbf{n}_a = 0, \tag{8.21}$$

which implies that when taking the sum over all surrounding elements, the first term on the RHS of (8.20) vanishes. The remaining two terms are the same as those obtained when performing a FVM line-integral approximation around node a.

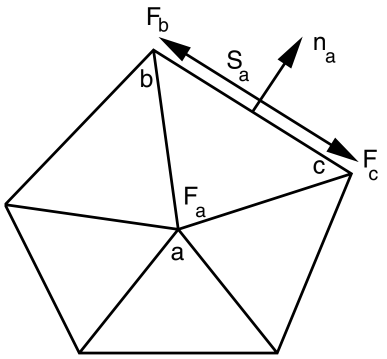


Figure 8.2. Equivalency of linear GFEM and FVM

The same equivalency between the GFEM with linear elements and FVMs can also be shown for 3-D tetrahedra.

8.2. Lax–Wendroff (Taylor–Galerkin)

All Lax–Wendroff type schemes are derived by performing a Taylor-series expansion in time (Lax and Wendroff (1960), Donea (1984), Löhner *et al.* (1984))

$$\Delta \mathbf{u} = \Delta t \mathbf{u}_{,t} + \frac{\Delta t^2}{2} \mathbf{u}_{,tt} \bigg|^{n+\Theta} . \tag{8.22}$$

Then the original equation,

$$\mathbf{u}_{,t} = -\nabla \cdot \mathbf{F}, \tag{8.23}$$

is used repeatedly in conjunction with the Jacobians of the fluxes $\mathcal{A}$,

$$\Delta\mathbf{F} = \mathcal{A} \cdot \Delta\mathbf{u}. \tag{8.24}$$

This results in

$$\Delta\mathbf{u} = -\Delta t \nabla \cdot \mathbf{F} + \frac{\Delta t^2}{2} \nabla \cdot \mathcal{A} \cdot \nabla \cdot \mathbf{F}\Big|^{n+\Theta}. \tag{8.25}$$

Linearization of the second term on the RHS via

$$\mathbf{F}|^{n+\Theta} = \mathbf{F}^n + \Theta\mathcal{A} \cdot \Delta\mathbf{u} \tag{8.26}$$

yields the final scheme:

$$\left[1 - \Theta\frac{\Delta t^2}{2} \nabla \cdot \mathcal{A} \otimes \mathcal{A}\nabla \cdot \right]\Delta\mathbf{u} = -\Delta t \nabla \cdot \mathbf{F} + \frac{\Delta t^2}{2} \nabla \cdot \mathcal{A} \cdot \nabla \cdot \mathbf{F}. \tag{8.27}$$

This timestepping scheme is then discretized in space using the GFEM. Note that the last term in (8.27) provides a matrix dissipation (or streamline upwinding for the scalar advection equation). This is very similar to the streamline upwind Petrov–Galerkin (SUPG) techniques (Tezduyar and Hughes (1983), Hughes and Tezduyar (1984), Le Beau and Tezduyar (1991)) for linear/bilinear elements, except that in the latter case the multiplicative factor is not the timestep but a mesh and flow variables dependent variable.

8.2.1. EXPEDITING THE RHS EVALUATION

The appearance of the Jacobians $\mathcal{A}$ makes the evaluation of the RHS tedious. Substantial savings may be realized by developing two-step schemes that approximate the RHS without recourse to $\mathcal{A}$ (Burstein (1967), Lapidus (1967), MacCormack (1969)). The simplest two-step scheme that accomplishes this goal is given by:

Half-step: predict $\mathbf{u}^{n+0.5}$

$$\mathbf{u}^{n+0.5} = \mathbf{u} - \frac{\Delta t}{2} \nabla \cdot \mathbf{F}. \tag{8.28a}$$

Full-step: use $\mathbf{u}^{n+0.5}$

$$\begin{aligned}\Delta\mathbf{u} &= -\Delta t \nabla \cdot \mathbf{F}(\mathbf{u}^{n+0.5}) = -\Delta t \nabla \cdot \mathbf{F}\left(\mathbf{u} - \frac{\Delta t}{2} \nabla \cdot \mathbf{F}\right) \\ &= -\Delta t \nabla \cdot \mathbf{F}(\mathbf{u}) + \frac{\Delta t^2}{2} \nabla \cdot \mathcal{A} \cdot \nabla \cdot \mathbf{F}(\mathbf{u}).\end{aligned} \tag{8.28b}$$

Several possibilities exist for the spatial discretization. If one chooses linear shape functions for both half-steps (N^i, N^i), a five-point stencil is obtained in one dimension. The scheme obtained is identical to a two-step Runge–Kutta GFEM. Thus, for the steady state no damping is present. Another possibility is to choose constant shape functions for the half-step solution (P^e, N^i). This choice recovers the original three-point stencil in one dimension, and for the linear advection equation yields the same scheme as that given by (8.27). Observe that this scheme has a second-order damping operator for the steady state. Thus, this second scheme requires no additional stabilization operator for 'mild' problems (e.g. subsonic flows without shocks). The difference between these schemes is shown conceptually in Figure 8.3.

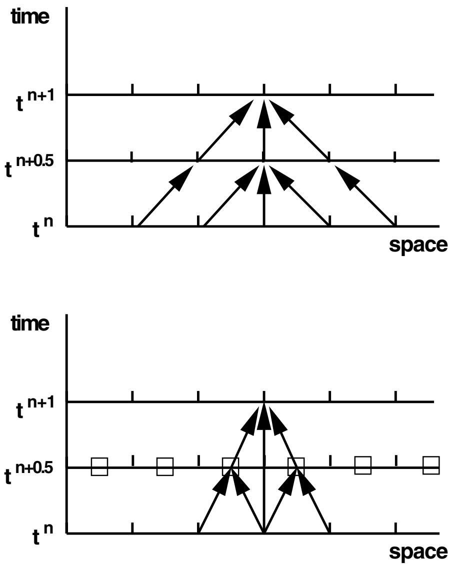


Figure 8.3. Two-step schemes

8.2.2. LINEAR ELEMENTS (TRIANGLES, TETRAHEDRA)

Because of the widespread use of the latter scheme, a more detailed description of it will be given. The spatial discretization is given by

- at $t^{n+\frac{1}{2}} = t^n + \frac{1}{2}\Delta t$: $\mathbf{u}$, $\mathbf{F}$ piecewise constant;
- at t^n, t^{n+1}: $\mathbf{u}$, $\mathbf{F}$ piecewise linear.

(a) *First step.* When written out, the first step results in the following `GATHER, ADD` operations:

$$\mathbf{u}_{el}^{n+\frac{1}{2}} = \frac{1}{Nn} \cdot \sum_{i=1}^{Nn} \mathbf{u}_i^n - \frac{\Delta t}{2} \cdot \sum_{i=1}^{Nn} N_{,k}^i F_i^k, \tag{8.29a}$$

where *Nn* denotes the number of nodes of the element.

(b) *Second step.* The second step involves `SCATTER-ADD` operations:

$$\int N^i N^j \, d\Omega \cdot \Delta \mathbf{u}_j = \frac{\Delta t}{Nn} \cdot \sum_{el} VOL_{el} \cdot N_{,k}^i F_{el}^k \bigg|^{n+\frac{1}{2}}, \tag{8.29b}$$

and results in the matrix system

$$\mathbf{M}_c \cdot \Delta \mathbf{u}^n = \mathbf{r}^n. \tag{8.30}$$

The flow of information for these two steps is shown in Figure 8.4.

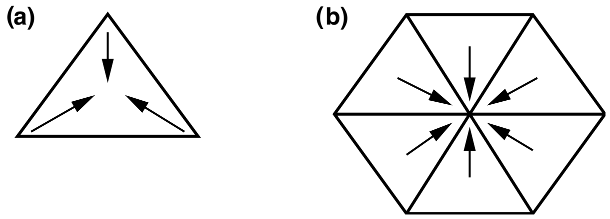


Figure 8.4. Two-step Taylor–Galerkin scheme

8.3. Solving for the consistent mass matrix

Due to its very favourable conditioning, (8.30) is best solved iteratively as

$$\mathbf{M}_l \cdot (\Delta \mathbf{u}^n_{i+1} - \Delta \mathbf{u}^n_i) = \mathbf{r}^n - \mathbf{M}_c \cdot \Delta \mathbf{u}^n, \quad i = 0, \ldots, niter, \quad \Delta \mathbf{u}^n_0 = 0. \tag{8.31}$$

In practice, no more than three iterations are employed ($niter \leq 3$).

8.4. Artificial viscosities

An observation made time and time again is that for shocks or other discontinuities the damping provided by the stabilized GFEM or the Lax–Wendroff schemes is not sufficient. The simulated flowfields exhibit unphysical results, with overshoots or undershoots in pressure, density, etc. In most cases, the numerical results will eventually diverge or, to use the more common term, ‘blow up’. Therefore, some additional artificial dissipation or stabilization (a more elegant term for the same thing) has to be added. In a real flow, the vicinity of a shock is controlled by the viscous effects that become large over the small width of the shock. The shock width is typically orders of magnitude smaller than the smallest mesh size used in current simulations. Therefore, this viscosity effect has to be scaled to the current mesh size. At the same time, the effects of these artificial viscosity terms should only be confined to shock or discontinuity regions. Thus, a sensing function must be provided to locate these regions in space. These considerations lead to artificial viscosity operators of the form

$$\mathbf{d} = \frac{\Delta t}{\Delta t_l} \nabla (h^2 f(\mathbf{u})) \nabla \mathbf{u}. \tag{8.32}$$

Here $f(\mathbf{u})$ denotes the sensing function and h the element size. The ratio of timestep taken (Δt) to allowable timestep (Δt_l) is necessary to avoid a change in solution when shifting from local to global timesteps. Some popular artificial viscosities are listed below.

(a) *Lapidus.* Defined by (see Figure 8.5)

$$\mathbf{d} = \Delta t \cdot \frac{\partial}{\partial l}\left(|k^{ll}| \frac{\partial \mathbf{u}}{\partial l} \right), \tag{8.33}$$

where

$$\mathbf{l} = \frac{\nabla |\mathbf{v}|}{|\nabla |\mathbf{v}||}, \quad k^{ll} = c_1 \cdot h^2 \cdot \frac{\partial (\mathbf{v} \cdot \mathbf{l})}{\partial l}. \tag{8.34a, b}$$

The salient features of this type of artificial viscosity are as follows:

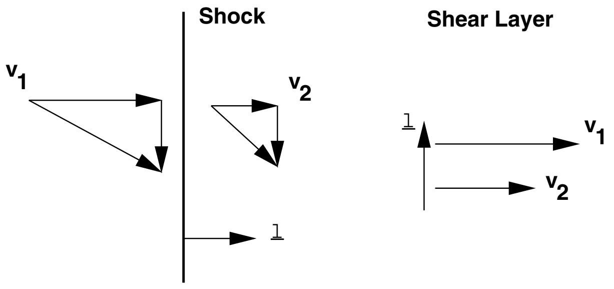


Figure 8.5. Lapidus artificial viscosity

- it is invariant under coordinate rotation;
- it is essentially 1-D and thus fast;
- it produces the desired effects at shocks;
- k^{ll} vanishes at shear and boundary layers;
- the identification of shocks ($k^{ll} < 0$) or expansions ($k^{ll} > 0$) is simple.

The Lapidus artificial viscosity (Lapidus (1967), Löhner *et al.* (1985a)) is most often employed for transient simulations in conjunction with more sophisticated FCT (Löhner *et al.* (1987)) or TVD schemes (Woodward and Colella (1984)) in order to add some background damping.

(b) *Pressure-based.* Given by

$$\mathbf{d} = \frac{\Delta t}{\Delta t_l} \nabla (h^2 f(p)) \nabla \mathbf{u}. \tag{8.35}$$

The salient features of this type of artificial viscosity are as follows:

- it is invariant under coordinate rotation;
- ∇^2 may be approximated by $(\mathbf{M}_l - \mathbf{M}_c)$, yielding a fast scheme;
- because the pressure p is near constant in shear and boundary layers, $f(p)$ should vanish there;
- usually, the enthalpy is taken for the energy equation in $\mathbf{u}$.

Many variations have been proposed for the sensing function $f(p)$. The three more popular ones are as follows:

(b1) *Jameson–Schmidt–Turkel*:

$$f(p) = c_1 \frac{|\nabla h^2 \nabla p|}{\bar{p}}, \tag{8.36}$$

which is very good for transonic flows (Jameson *et al.* (1981), Löhner *et al.* (1985b));

(b2) *Hassan–Morgan–Peraire*:

$$f(p) = c_2 \frac{|\nabla h^2 \nabla p|}{|h\nabla p|}, \tag{8.37}$$

which is more suitable for hypersonic flows (Hassan *et al.* (1990)); and

(b3) *Swanson–Turkel*:

$$f(p) = c_3 \frac{|\nabla h^2 \nabla p|}{\alpha|h\nabla p| + (1-\alpha)\bar{p}}, \tag{8.38}$$

which is a blend of the first two (Swanson and Turkel (1992)). Typically, $\alpha = 0.5$.

8.5. Boundary conditions

No numerical scheme is complete without proper ways of imposing boundary conditions. The Euler equations represent a hyperbolic system of PDEs. Linearization around any arbitrary state yields a system of advection equations for so-called characteristic variables (Usab and Murman (1983), Thomas and Salas (1985)). The proper handling of these boundary conditions will be discussed below. The following notation will be employed:

ρ:	the density;
v_n:	the normal velocity component at the boundary (pointing inwards);
v_t:	the tangential velocity component at the boundary;
p:	the pressure; and
c:	the velocity of sound.

Furthermore, use will be made of the Bernoulli equation, given by the total pressure relation:

$$\frac{\gamma}{\gamma-1}\frac{p_0}{\rho_0} + \frac{v_0^2}{2} = \frac{\gamma}{\gamma-1}\frac{p}{\rho} + \frac{v^2}{2} = \text{const.}, \tag{8.40}$$

and the isentropic relation

$$\frac{p}{\rho^\gamma} = \frac{p_0}{\rho_0^\gamma}. \tag{8.41}$$

Given a velocity, and assuming constant total pressure along the streamlines, the pressure and density may be computed as

$$\frac{p}{\rho} = \frac{p_0}{\rho_0} + \frac{\gamma-1}{2\gamma}[v_0^2 - v^2], \tag{8.42}$$

and from the isentropic relation (8.41),

$$\rho = \left[\frac{\rho_0^\gamma}{p_0}\frac{p}{\rho}\right]^{1/(\gamma-1)}. \tag{8.43}$$

With ρ, p is obtained again from (8.41):

$$p = \frac{p_0}{\rho_0^\gamma}\rho^\gamma. \tag{8.44}$$

Suppose that an explicit evaluation of the fluxes (i.e. the RHS) has been performed and the unknowns have been updated. These predicted unknowns, that have to be corrected at boundaries, will be denoted by '$*$', e.g. the predicted density as ρ_*. A linearized characteristics analysis can be used to correct the predicted ($*$) values; performing a 1-D analysis at the boundary, in the direction normal to it, we have, with the *inward* normal **n** (see Figure 8.6), the following set of eigenvalues and eigenvectors:

$$\lambda = \begin{pmatrix} v_n \\ v_n \\ v_n + c \\ v_n - c \end{pmatrix}, \quad \mathbf{W} = \begin{pmatrix} \rho - p/\overline{c}^2 \\ v_t \\ [v_n + p/(\overline{\rho c})] \\ [-v_n + p/(\overline{\rho c})] \end{pmatrix}, \tag{8.45}$$

where $\bar{u}$ denotes the average between the values at the previous timestep and the predicted state for unknown u.

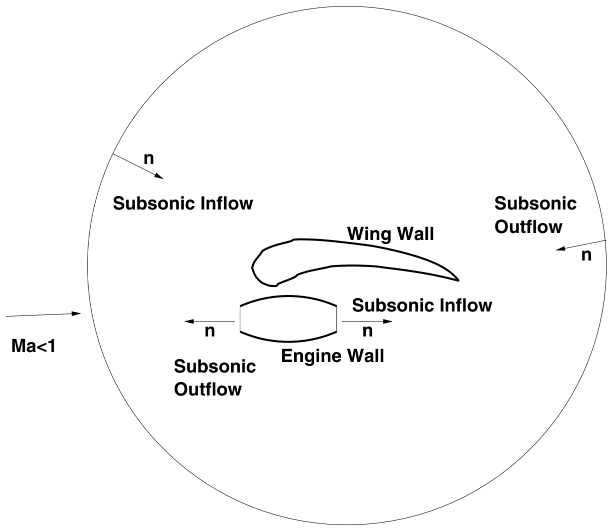


Figure 8.6. Boundary conditions for wing

One is now in a position to correct the predicted variables depending on the Mach number of the flow. The possible states are: supersonic inflow, subsonic inflow, subsonic outflow and supersonic outflow. Let us consider them in turn.

(a) *Supersonic inflow.* In this case, there should be no change in the variables, as no information can propagate upstream. Therefore, the variables are reset to the values at the beginning of the timestep. Thus, the predicted values are discarded in this case.

(b) *Subsonic inflow.* The incoming characteristics are λ_{1-3}, while the outgoing characteristic is λ_4. The values for the variables W_1, W_2, W_3, corresponding to the incoming characteristics, are taken from the 'state at infinity', and W_4 is taken from the predicted state. This results in

$$\begin{aligned} \rho - p/\overline{c}^2 &= \rho_\infty - p_\infty/\overline{c}^2, \\ v_t &= v_{t\infty}, \\ v_n + p/(\overline{\rho c}) &= v_{n\infty} + p_\infty/(\overline{\rho c}), \\ -v_n + p/(\overline{\rho c}) &= -v_{n*} + p_*/(\overline{\rho c}) \end{aligned} \tag{8.46}$$

or, after elimination,

$$\begin{aligned} &v_t = v_{t\infty}; \quad p = 0.5[p_\infty + p_* + \overline{\rho c}(v_{n\infty} - v_{n*})] \\ &\to \rho = \rho_\infty + (p - p_\infty)/\overline{c}^2; \quad v_n = v_{n\infty} + (p_\infty - p)/(\overline{\rho c}). \end{aligned} \tag{8.47}$$

For the case of *total* pressure one first computes, from the predicted velocity $\mathbf{v}_*$ the corresponding pressures and densities. These pressures and densities are then taken as the new 'state at infinity'. The resulting equations are

$$\begin{aligned} &v_t = v_{t\infty}; \quad v_n = v_{n*} \\ &\to p = 0.5(p_\infty + p_*); \quad \to \rho = \rho_\infty + (p - p_\infty)/\overline{c}^2. \end{aligned} \tag{8.48}$$

(c) *Subsonic outflow.* The incoming characteristics are λ_3, while the outgoing characteristics are $\lambda_{1,2,4}$. The value for the variable W_3, corresponding to the incoming characteristic, is taken from the 'state at infinity', and W_1, W_2, W_4 are taken from the predicted state. The resulting equations are

$$\begin{aligned} \rho - p/\overline{c}^2 &= \rho_* - p_*/\overline{c}^2, \\ v_t &= v_{t*}, \\ v_n + p/(\overline{\rho c}) &= v_{n\infty} + p_\infty/(\overline{\rho c}), \\ -v_n + p/(\overline{\rho c}) &= -v_{n*} + p_*/(\overline{\rho c}). \end{aligned} \tag{8.49}$$

There are several possibilities, depending on what has to be specified.

(c1) Prescribed state $v_{n\infty}$, p_∞:

$$\begin{aligned} &v_t = v_{t*}; \quad p = 0.5[p_\infty + p_* + \overline{\rho c}(v_{n\infty} - v_{n*})] \\ &\to \rho = \rho_* + (p - p_*)/\overline{c}^2; \quad v_n = v_{n\infty} + (p_\infty - p)/(\overline{\rho c}). \end{aligned} \tag{8.50}$$

(c2) Prescribed pressure p_∞:

$$\begin{aligned} &v_t = v_{t*}; \quad p = p_\infty \\ &\to \rho = \rho_* + (p_\infty - p_*)/\overline{c}^2; \quad v_n = v_{n*} + (p_\infty - p_*)/(\overline{\rho c}). \end{aligned} \tag{8.51}$$

(c3) Prescribed Mach number m_∞:

$$\begin{aligned} &v_t = v_{t*}; \quad v_n = -c_* m_\infty \\ &\to p = p_* + \rho_* c_*(v_n - v_{n*}); \quad \to \rho = \rho_* + (p - p_*)/\overline{c}^2. \end{aligned} \tag{8.52}$$

(c4) Prescribed mass flux $(\rho v_n)_\infty$:

$$\begin{aligned} v_t = v_{t*};\ p = p_* + c_*((\rho v_n)_\infty - \rho_* v_{n*}) \\ \to \rho = \rho_* + (p - p_*)/\bar{c}^2;\ \to v_n = (\rho v_n)_\infty/\rho. \end{aligned} \tag{8.53}$$

(d) *Supersonic outflow.* In this case, no information can propagate into the computational domain from the exterior. Therefore, the predicted variables are left unchanged.

(e) *Solid walls.* For walls, the only boundary condition one can impose is the no-penetration condition, taking into consideration the wall velocity **w**. The predicted momentum at the surface $\Delta\rho\mathbf{v}_*$ is decomposed as

$$\Delta\rho\mathbf{v}_* = \Delta[\rho(\mathbf{w} + \alpha\mathbf{t} + \beta\mathbf{n})], \tag{8.54}$$

where **t** and **n** are the tangential and normal vectors, respectively. The desired momentum at the new timestep should, however, have no normal velocity component ($\beta = 0$) and it has the form

$$\Delta\rho\mathbf{v}^{n+1} = \Delta[\rho(\mathbf{w} + \alpha\mathbf{t})]. \tag{8.55}$$

Combining the last two equations, we obtain the two following cases.

(e1) Given **t**:

$$\Delta\rho\mathbf{v}^{n+1} = \Delta\rho\mathbf{w} + [(\Delta\rho\mathbf{v}_* - \Delta\rho\mathbf{w})\cdot\mathbf{t}]\cdot\mathbf{t}. \tag{8.56}$$

(e2) Given **n**:

$$\Delta\rho\mathbf{v}^{n+1} = \Delta\rho\mathbf{v}_* - [(\Delta\rho\mathbf{v}_* - \Delta\rho\mathbf{w})\cdot\mathbf{n}]\cdot\mathbf{n}. \tag{8.57}$$

(f) *Porous walls.* For porous walls, the velocity normal to the wall is given as a function of the difference $\Delta p = p_i - p_o$ between the inner and outer pressures. This function is obtained from measurements. A typical way to model the behaviour is via a linear–linear model of the form

$$|\Delta p| < p_2 : v_n = c_1 \frac{\Delta p}{p_1}, \tag{8.58}$$

$$|\Delta p| \geq p_2 : v_n = c_1 \frac{p_b}{p_1} + c_2 \frac{\Delta p - p_b}{p_1}, \tag{8.59}$$

where c_1, c_2, p_1, p_2 are empirically determined constants.

8.6. Viscous fluxes

After schemes that are suitable for the Euler equations have been described, the proper discretization of the viscous fluxes must be considered. The RHS for the Galerkin weighted residual statement is of the form

$$\mathbf{r}^i = \int N^k \mathbf{F}^v \, d\Omega. \tag{8.60}$$

As the viscous and thermal fluxes are themselves functions of the flow variables that involve derivatives (equations (8.2) and (8.4)), one must either recover them at points or at the element level. The first possibility results in two passes over the mesh as follows.

(a1) Evaluation of viscous fluxes at points, e.g.,

$$\int N^k N^l \sigma_{ij}\, d\Omega = \int N^k \left[\mu \left(\frac{\partial v_i}{\partial x_j} + \frac{\partial v_j}{\partial x_i} \right) + \lambda \frac{\partial v_k}{\partial x_k} \delta_{ij} \right] d\Omega, \tag{8.61}$$

which may be recast as

$$\mathbf{M}_c \sigma_{ij} = \int N^k \left[\mu \left(\frac{\partial N^l}{\partial x_j} \hat{v}_i|_l + \frac{\partial N^l}{\partial x_i} \hat{v}_j|_l \right) + \lambda \frac{\partial N^l}{\partial x_k} \delta_{ij} \hat{v}_k|_l \right] d\Omega. \tag{8.62}$$

(a2) Given the stresses σ_{ij} at the nodes, evaluation of viscous flux derivatives to form the RHS:

$$\mathrm{RHS}^{ik} = \int N^k \frac{\partial \sigma_{ij}}{\partial x_j}\, d\Omega = \int N^k \frac{\partial N^l}{\partial x_j}\, d\Omega\, \sigma_{ij}. \tag{8.63}$$

The second possibility, which is more natural for FEMs, results in

$$\mathrm{RHS}^{ik} = \int N^k \frac{\partial \sigma_{ij}}{\partial x_j}\, d\Omega = -\int \frac{\partial N^k}{\partial x_j} \sigma_{ij}\, d\Omega + \int n_j N^k \sigma_{ij}\, d\Gamma. \tag{8.64}$$

Here n_j denotes the component of the surface normal vector in the jth direction. Inserting the appropriate expressions for the stresses results in

$$\begin{aligned} \mathrm{RHS}^{ik} = &-\int \frac{\partial N^k}{\partial x_j} \left[\mu \left(\frac{\partial N^l}{\partial x_j} \hat{v}_i|_l + \frac{\partial N^l}{\partial x_i} \hat{v}_j|_l \right) + \lambda \frac{\partial N^l}{\partial x_k} \delta_{ij} \hat{v}_k|_l \right] d\Omega \\ &+ \int n_j N^k \left[\mu \left(\frac{\partial N^l}{\partial x_j} \hat{v}_i|_l + \frac{\partial N^l}{\partial x_i} \hat{v}_j|_l \right) + \lambda \frac{\partial N^l}{\partial x_k} \delta_{ij} \hat{v}_k|_l \right] d\Gamma. \end{aligned} \tag{8.65}$$

If one evaluates these expressions within a simple 1-D context, it can be observed that the first possibility results in a five-point stencil, whereas the second possibility results in a more compact stencil of only three points. The second possibility is clearly better suited for element-based codes, as it represents the natural, self-adjoint discretization of the viscous stresses and thermal fluxes. For edge-based solvers the additional storage required for all the i, j components of the tensor of second derivatives would more than double the memory requirements. For this reason, most edge-based solvers use the first form when evaluating the viscous terms.

A third way to evaluate the viscous fluxes is found by separating the terms forming Laplacian operators from the rest of the viscous fluxes. For the momentum equations this results in

$$\frac{\partial \sigma_{ij}}{\partial x_j} = \frac{\partial}{\partial x_j} \left(\mu \frac{\partial v_i}{\partial x_j} \right) + \frac{\partial}{\partial x_j} \left(\mu \frac{\partial v_j}{\partial x_i} + \lambda \frac{\partial v_k}{\partial x_k} \delta_{ij} \right), \tag{8.66}$$

and for the energy equation

$$\frac{\partial}{\partial x_j} \left(\frac{v_l \sigma_{lj} + k \partial T}{\partial x_j} \right) = \frac{\partial}{\partial x_j} \left(\frac{\mu}{2} \frac{\partial v_l v_l}{\partial x_j} + k \frac{\partial T}{\partial x_j} \right) + \frac{\partial}{\partial x_j} \left(\mu v_l \frac{\partial v_j}{\partial x_l} + \lambda \frac{\partial v_k}{\partial x_k} \delta_{lj} \right). \tag{8.67}$$

Unless pronounced compressibility effects are present, the dominant terms are the ones that result in purely Laplacian operators. These are then treated in the usual FEM way by storing only one additional entry (for the Laplacian) per edge. The remainder terms are treated by taking first derivatives of gradients. For most solvers the gradients are required for limiting, so there is almost no extra cost involved.

9 FLUX-CORRECTED TRANSPORT SCHEMES

Although flux-corrected transport (FCT) schemes have not enjoyed the popularity of total variation diminishing (TVD) schemes in the steady-state CFD community, they are included here for the following reasons.

(a) FCT schemes were the first to introduce the concept of limiters (Boris and Book (1973, 1976), Book and Boris (1975)), and particularly for didactic purposes it never hurts to go back to the source.

(b) FCT schemes are used almost exclusively in the unsteady CFD and plasma physics communities.

(c) Unlike TVD schemes, FCT limiting is based directly on the unknowns chosen. This makes FCT ideally suited to introduce the concept of nonlinear schemes and limiting.

FCT schemes were developed from the observation that linear schemes, however good, perform poorly for transient problems with steep gradients. For these linear schemes, the choice is between high-order, oscillatory, 'noisy' solutions, or low-order, overdiffusive, 'smooth' solutions. Godunov's theorem (Godunov (1959)) states this same observation as:

Theorem. (Godunov) *No linear scheme of order greater than 1 will yield monotonic (wiggle-free, ripple-free) solutions.*

The way to circumvent this stringent theorem is to develop *nonlinear schemes*. This is most easily accomplished by mixing a high-order and a low-order scheme. The process of combining these two schemes in a rational way is called *limiting*.

Any high-order scheme used to advance the solution either in time or between iterations towards the steady state may be written as

$$u^{n+1} = u^n + \Delta\mathbf{u}^h = u^n + \Delta\mathbf{u}^l + (\Delta\mathbf{u}^h - \Delta\mathbf{u}^l) = u^l + (\Delta\mathbf{u}^h - \Delta\mathbf{u}^l). \tag{9.1}$$

Here $\Delta\mathbf{u}^h$ and $\Delta\mathbf{u}^l$ denote the increments obtained by the high- and low-order scheme, respectively, and u^l is the monotone, ripple-free solution at time $t = t^{n+1}$ of the low-order scheme. The idea behind FCT is to limit the second term on the RHS of (9.1),

$$u^{n+1} = u^l + \lim(\Delta\mathbf{u}^h - \Delta\mathbf{u}^l), \tag{9.2}$$

in such a way that no new over/undershoots are created. It is at this point that a further constraint, given by the original conservation law itself, must be taken into account: strict conservation on the discrete level should be maintained. This means that the limiting process will have to be carried out wherever RHS contributions are computed: for element-based schemes at the element level, for edge-based schemes at the edges.

9.1. Algorithmic implementation

The concept of FCT can now be translated into a numerical scheme. The description is given for element-based schemes. For edge-based schemes, replace ‘element’ by ‘edge’ in what follows. For finite volume schemes, one may replace ‘element’ by ‘cell’. FCT consists of the following six algorithmic steps:

1. Compute *LEC*: the element contributions from some low-order scheme guaranteed to give monotonic results for the problem at hand;

2. Compute *HEC*: the element contributions given by some high-order scheme;

3. Define the anti-diffusive element contributions:

$$AEC = HEC - LEC; \tag{9.3}$$

4. Compute the updated low-order solution u^l:

$$u_i^l = u_i^n + \sum_{el} LEC, \quad i = 1, \ldots, npoin; \tag{9.4}$$

5. Limit or correct the *AEC* so that u^{n+1} as computed in step 6 is free of extrema not also found in u^l or u^n:

$$AEC^c = C_{el} \cdot AEC, \quad 0 \leq C_{el} \leq 1; \tag{9.5}$$

6. Apply the limited *AEC*:

$$u_i^{n+1} = u_i^l + \sum_{el} AEC^c. \tag{9.6}$$

9.1.1. THE LIMITING PROCEDURE

Obviously, the whole approach depends critically on step 5 above. If we consider an isolated point surrounded by elements, the task of the limiting procedure is to insure that the increments or decrements due to the anti-diffusive element contributions AEC do not exceed a prescribed tolerance (see Figure 9.1).

In the most general case, the contributions to a point will be a mix of positive and negative contributions. Given that the *AEC*s will be limited, i.e. multiplied by a number $0 \leq C_{el} \leq 1$, it may happen that after limiting all positive or negative contributions vanish. The largest increment (decrement) will occur when only the positive (negative) contributions are considered. For this reason, we must consider what happens if only positive or only negative contributions are added to a point. The comparison of the allowable increments and decrements with these all-positive and all-negative contributions then yields the maximum allowable percentage of the AECs that may be added or subtracted to a point. On the other hand, an element may contribute to a number of nodes and, in order to maintain strict conservation, the limiting must be performed for all the element node contributions in the same way. Therefore, a comparison for all the nodes of an element is performed, and the smallest of the evaluated percentages that applies is retained.

Define the following quantities:

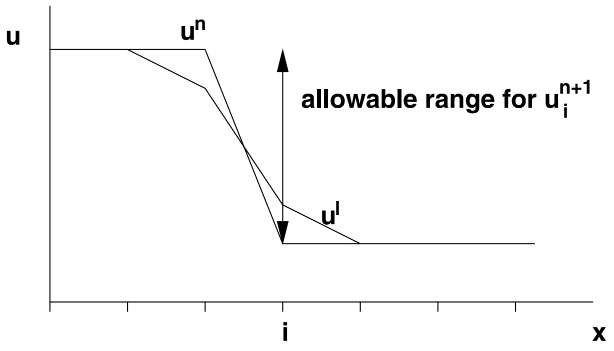


Figure 9.1. Limiting procedure

$P_i^{\pm}$: the sum of all positive (negative) element contributions to node i,

$$P_i^{\pm} = \sum_{el} \begin{Bmatrix} \max \\ \min \end{Bmatrix} (0, AEC_{el}); \tag{9.7}$$

$Q_i^{\pm}$: the maximum (minimum) increment node i is allowed to achieve in step 6 above,

$$Q_i^{\pm} = u_i^{\overset{\max}{\min}} - u^l. \tag{9.8}$$

The ratio of positive and negative contributions that ensure monotonicity is then given by

$$R^{\pm} := \begin{cases} \min(1,\ Q^{\pm}/P^{\pm}) & P^{+} > 0 > P^{-}, \\ 0 & \text{otherwise}. \end{cases} \tag{9.9}$$

For the elements, the final value taken is the most conservative:

$$C_{el} = \min(\text{element nodes}) \begin{cases} R^{+} & \text{if } AEC > 0, \\ R^{-} & \text{if } AEC < 0. \end{cases} \tag{9.10}$$

The allowed value $u_i^{\overset{\max}{\min}}$ is taken between each point and its nearest-neighbours. For element-based schemes, it may be obtained in three steps as follows:

(a) maximum (minimum) nodal unknowns of u^n and u^l:

$$u_i^{*} = \begin{Bmatrix} \max \\ \min \end{Bmatrix} (u_i^l, u_i^n); \tag{9.11}$$

(b) maximum (minimum) nodal value of element:

$$u_{el}^{*} = \begin{Bmatrix} \max \\ \min \end{Bmatrix} (u_A^{*}, u_B^{*}, \ldots, u_C^{*}); \tag{9.12}$$

(c) maximum (minimum) unknowns of all elements surrounding node i:

$$u_i^{\overset{\max}{\min}} = \begin{Bmatrix} \max \\ \min \end{Bmatrix} (u_1^*, u_2^*, \ldots, u_m^*). \tag{9.13}$$

A number of variations are possible for $u_i^{\overset{\max}{\min}}$. For example, the so-called ‘clipping limiter’ is obtained by setting $u_i^* = \left\{\begin{smallmatrix} \max \\ \min \end{smallmatrix}\right\} u_i^l$ in (a), i.e. by not looking back to the solution at the previous timestep or iteration, but simply comparing nearest-neighbour values at the new timestep or iteration for the low-order scheme. As remarked before, the limiting is based solely on the unknowns u, not on a ratio of differences as in most TVD schemes.

9.2. Steepening

The anti-diffusive step was designed to steepen the low-order solution obtained at the new timestep or iteration. In some cases, particularly for high-order schemes of order greater than two, the anti-diffusive step can flatten the profile of the solution even further, or lead to an increase of wiggles and noise. This is the case even though the solution remains within its allowed limits. A situation where this is the case can be seen from Figure 9.2.

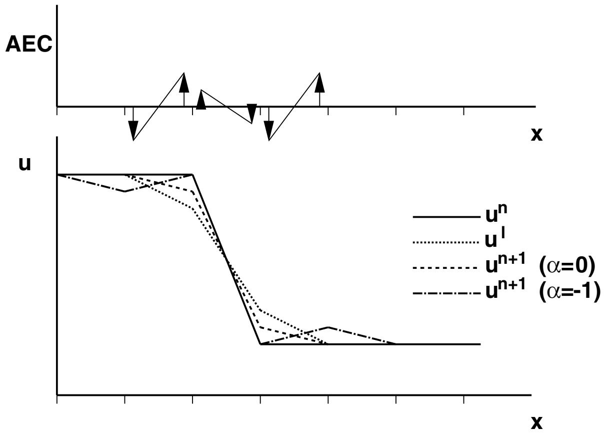


Figure 9.2. Steepener for FCT

This type of behaviour can be avoided if the anti-diffusive flux is either set to zero or reversed. A simple way to decide when to reverse the anti-diffusive fluxes is to compute the scalar product of the low-order solution at the new timestep or iteration and the anti-diffusive element contributions:

$$\nabla u^l \cdot AEC < 0 \Rightarrow AEC = -\alpha \cdot AEC, \tag{9.14}$$

with $0 < \alpha < 1$. Values of α greater than unity lead to steepening. This can be beneficial in some cases, but is highly dangerous, as it can lead to unphysical solutions (e.g. spurious contact discontinuities) in complex applications.

9.3. FCT for Taylor–Galerkin schemes

For the *explicit Taylor–Galerkin* schemes, FCT takes a particularly simple form. We have:

- high-order: consistent mass Taylor–Galerkin,

$$\mathbf{M}_c \cdot \Delta\mathbf{u}^h = \mathbf{r} \Rightarrow \mathbf{M}_l \cdot \Delta\mathbf{u}^h = \mathbf{r} + (\mathbf{M}_l - \mathbf{M}_c) \cdot \Delta\mathbf{u}^h; \tag{9.15a}$$

- low-order: lumped mass Taylor–Galerkin + diffusion,

$$\mathbf{M}_l \cdot \Delta\mathbf{u}^l = \mathbf{r} + \mathbf{d} = \mathbf{r} - c_\tau(\mathbf{M}_l - \mathbf{M}_c) \cdot \mathbf{u}, \tag{9.15b}$$

  where c_τ denotes a diffusion coefficient.

Thus

$$AEC = [(\mathbf{M}_l - \mathbf{M}_c) \cdot (c_\tau \mathbf{u} + \Delta\mathbf{u}^h)]_{el}, \tag{9.16}$$

implying that *no physical flux terms are left in AEC.* This is typical of FCT schemes, and reflects the fact that monotonicity may be imposed without knowledge of the physics described by the fluxes. This separation of numerics and physics has several advantages, among them speed and generality. The low-order solution will be monotone for a diffusion coefficient $c_\tau = 1$. For cases where a large variation of possible timesteps is encountered throughout the mesh, the allowable diffusion factor may be much lower. The ‘optimal’ choice (in the sense of least diffusive yet monotone low-order scheme) is to take $c_\tau = \Delta t_g/\Delta t_l$, where Δt_g is the global timestep chosen (i.e. the minimum over all elements) and Δt_l the allowable local timestep ($\Delta t_l \geq \Delta t_g$).

9.4. Iterative limiting

Given that the anti-diffusive element contributions surrounding a point can have different signs, and that a ‘most conservative’ compromise has to be reached for positive and negative contributions, the remaining (i.e. not yet added) anti-diffusive element contributions may still be added to the new solution without violating monotonicity constraints. This may be achieved by the following iterative limiting procedure:

For iterations $j = 1, k$:

- perform limiting procedure,

$$u^{n+1} = u^l + \sum_{el} C_{el} \cdot AEC;$$

- update remaining anti-diffusive element contributions,

$$AEC \leftarrow (1 - C_{el})AEC;$$

- re-define the low-order solution at t^{n+1},

$$u^l = u^{n+1}.$$

Experience indicates that, for explicit schemes, the improvements obtained by this iterative limiting procedure are modest. However, for implicit schemes, the gains are considerable and well worth the extra computational effort (Kuzmin (2001), Kuzmin and Turek (2002), Kuzmin *et al.* (2003, 2005)).

9.5. Limiting for systems of equations

The results available in the literature (Boris and Book (1973), Book and Boris (1975), Boris and Book (1976), Zalesak (1979), Parrott and Christie (1986), Löhner *et al.* (1987, 1988), Zalesak and Löhner (1989)) indicate that with FCT results of excellent quality can be obtained for a single PDE, e.g. the scalar advection equation. However, in the attempt to extend the limiting process to systems of PDEs no immediately obvious or natural limiting procedure becomes apparent. Obviously, for 1-D problems one could advect each simple wave system separately, and then assemble the solution at the new timestep. However, for multi-dimensional problems such a splitting is not possible, as the acoustic waves are circular in nature. FDM–FCT codes used for production runs (Fry and Book (1983), Fyfe *et al.* (1985)) have so far limited each equation separately, invoking operator-splitting arguments. This approach does not always give very good results, as may be seen from Sod's (1978) comparison of schemes for the Riemann problem, and has been a point of continuing criticism by those who prefer to use the more costly Riemann-solver-based, essentially 1-D TVD schemes (van Leer (1974), Roe (1981), Osher and Solomon (1982), Harten (1983), Sweby (1984), Colella (1990), Woodward and Colella (1984)). An attractive alternative is to introduce 'system character' for the limiter by combining the limiters for all equations of the system. Many variations are possible and can be implemented, giving different performance for different problems. Some of the possibilities are listed here, with comments where empirical experience is available.

(a) *Independent treatment of each equation as in operator-split FCT.* This is the least diffusive method, tending to produce an excessive amount of ripples in the non-conserved quantities (and ultimately also in the conserved quantities).

(b) *Use of the same limiter* (C_{el}) *for all equations.* This produces much better results, seemingly because the phase errors for all equations are 'synchronized'. This was also observed by Harten and Zwaas (1972) and Zhmakin and Fursenko (1981) for a class of schemes very similar to FCT. We mention the following possibilities.

(i) Use of a certain variable as 'indicator variable' (e.g. density, pressure, entropy).

(ii) Use of the minimum of the limiters obtained for the density and the energy ($C_{el} = \min(C_{el}(\rho), C_{el}(\rho e))$): this produces acceptable results, although some undershoots for very strong shocks are present. This option is currently the preferred choice for strongly unsteady flows characterized by propagating and/or interacting shock waves.

(iii) Use of the minimum of the limiters obtained for the density and the pressure ($C_{el} = \min(C_{el}(\rho), C_{el}(p))$): this again produces acceptable results, particularly for steady-state problems.

9.5.1. LIMITING ANY SET OF QUANTITIES

A general algorithm to limit any set of quantities may be formulated as follows:

- define a new set of (non-conservative) variables $\mathbf{u}'$;
- transform $\mathbf{u}, \mathbf{u}^l \rightarrow \mathbf{u}', \mathbf{u}'^l$ and see how much $\mathbf{u}'$ can change at each point $\Rightarrow \Delta\mathbf{u}'|_{\min}^{\max}$;

- define a mean value of $\mathbf{u}^l$ in the elements and evaluate

$$\Delta\mathbf{u}' = \mathbf{A}(\overline{\mathbf{u}}^l) \cdot \Delta\mathbf{u}; \tag{9.17}$$

- limit the transformed increments $\Delta\mathbf{u}' \Rightarrow C'_{el} \Rightarrow \Delta\mathbf{u}'' = \lim(\Delta\mathbf{u}')$;

- transform back the variables and add

$$\Delta\mathbf{u}^* = \mathbf{A}^{-1}(\overline{\mathbf{u}}^l) \cdot \Delta\mathbf{u}'' = \mathbf{A}^{-1}(\overline{\mathbf{u}}^l) \cdot C'_{el} \cdot \mathbf{A}(\overline{\mathbf{u}}^l) \cdot \Delta\mathbf{u}. \tag{9.18}$$

Excellent results using limiting based on characteristic variables have been reported by Zalesak (2005) (see also Kuzmin *et al.* (2003, 2005)).

9.6. Examples

9.6.1. SHOCK TUBE

This is a classic example, which was used repeatedly to compare different Euler solvers (Sod (1978)). Initially, a membrane separates two fluid states given by $\rho_1 = 1.0$, $\mathbf{v}_1 = 0.0$, $p_1 = 1.0$ and $\rho_2 = 0.1$, $\mathbf{v}_2 = 0.0$, $p_2 = 0.1$. The membrane ruptures, giving rise to a shock, a contact discontinuity and a rarefaction wave. The results were obtained for an FCT run with limiter synchronization on the density and energy. Figure 9.3 shows the surface mesh and the surface contours of the density. A line cut through the 3-D mesh is compared to the exact solution in Figure 9.4. Note that the number of points appearing here corresponds to the faces of the tetrahedra being cut, i.e., it is two to three times the number of actual points. One can see that the shock and contact discontinuities are captured over two or four elements, respectively.

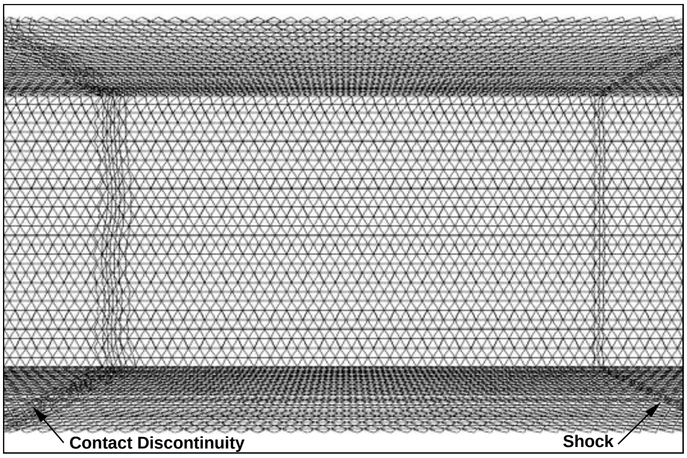


Figure 9.3. Shock tube: surface mesh and density

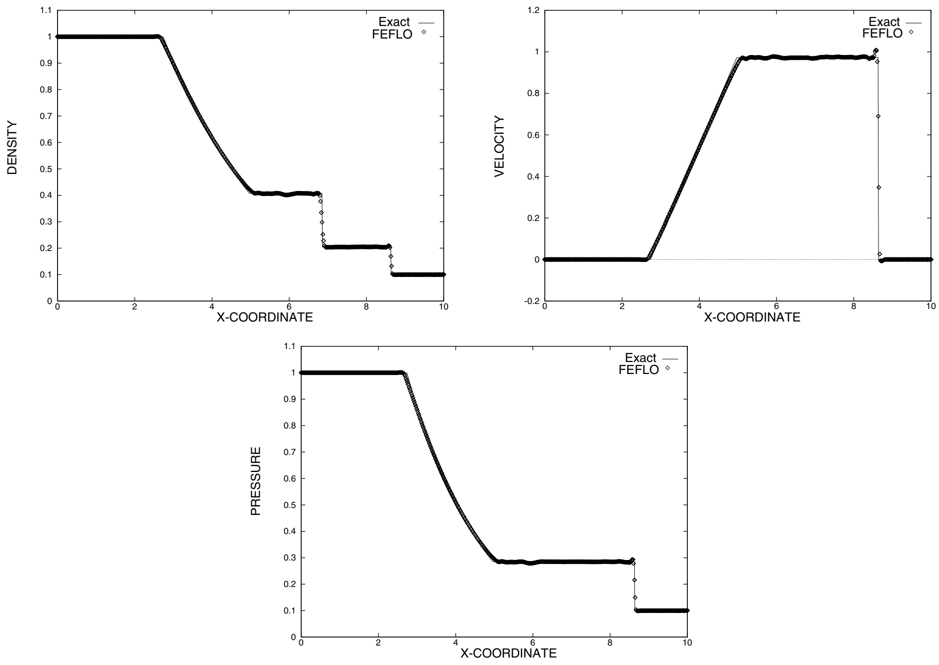


Figure 9.4. Shock tube: comparison to exact values

9.6.2. SHOCK DIFFRACTION OVER A WALL

The second example shown is typical of some of the large-scale blast simulations carried out with FCT schemes over the last decade (Baum and Löhner (1991), Sivier *et al.* (1992), Baum *et al.* (1993, 1995), Baum and Löhner (1994), Löhner *et al.* (1999)), and is taken from Rice *et al.* (2000). The outline of the domain is shown in Figure 9.5. Surface pressures for an axisymmetric and 3-D run, together with a comparison to experimental photographs are given in Figures 9.6 and 9.7. The comparison to experimental results at different stations is shown in Figure 9.8. As one can see, FCT schemes yield excellent results for this class of problem.

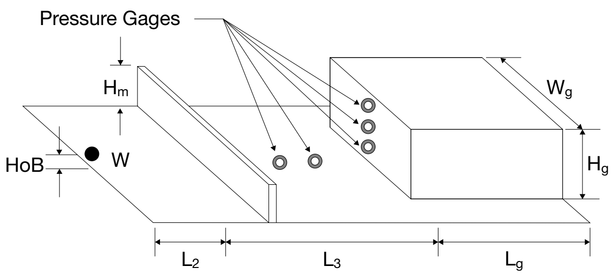


Figure 9.5. Shock diffraction over a wall

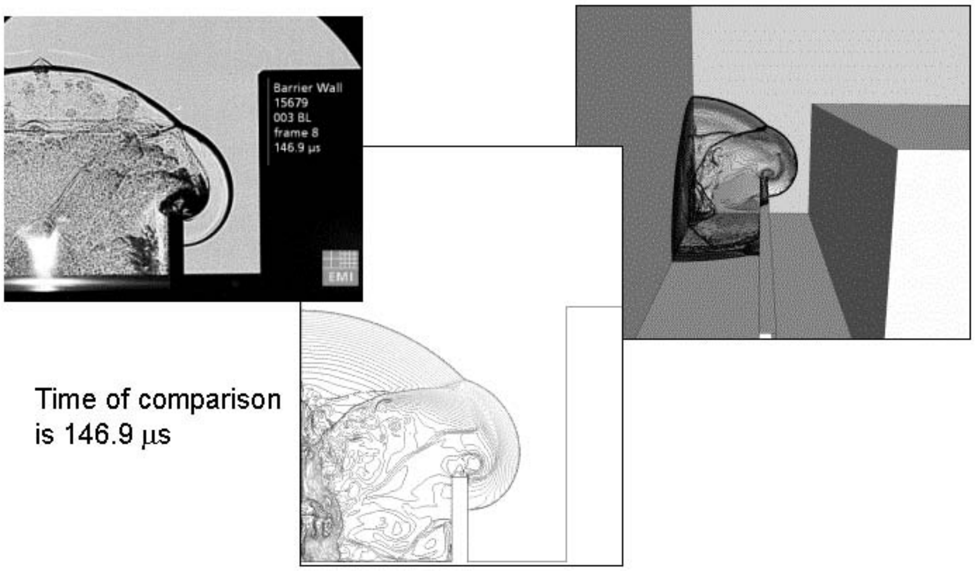


Figure 9.6. Results at time $t = 146\ \mu$s

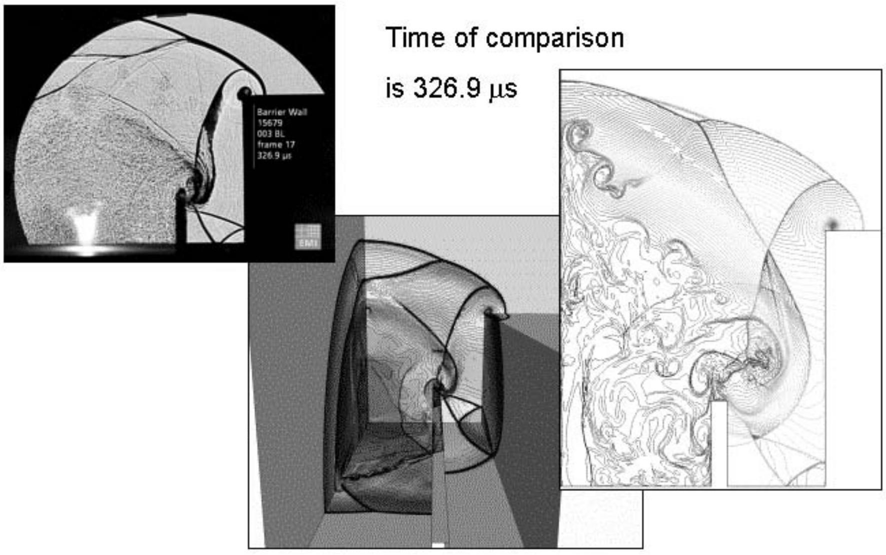


Figure 9.7. Results at time $t = 327\ \mu$s

9.7. Summary

The notion of limiting was introduced by reviewing FCT schemes. As seen above, these schemes do not require gradient evaluations for the limiting procedure, and in some instances

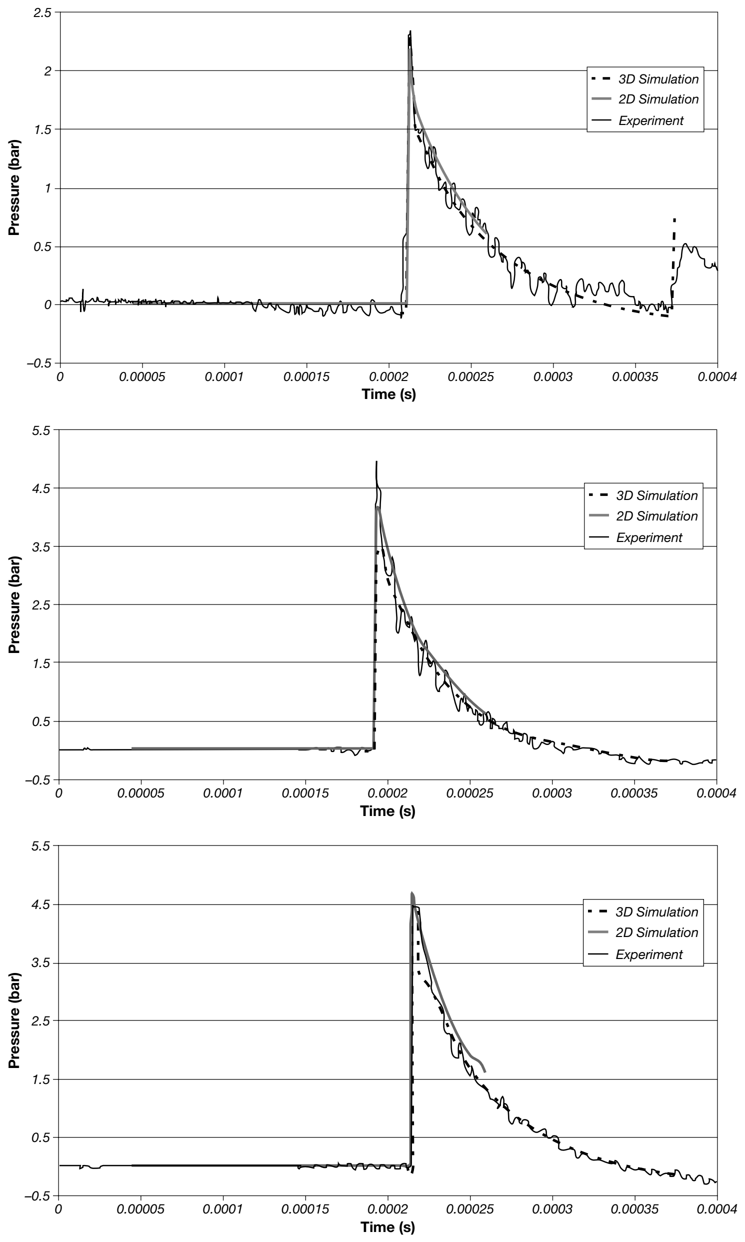


Figure 9.8. Comparison to experimental values

avoid the recalculation of fluxes while limiting. This makes them extremely fast as compared to TVD schemes. Due to their generality and ease of extension to other physical problems that require monotonicity preserving schemes, FCT has been used extensively in many fields, among them simulation of shock propagation (Fyfe *et al.* (1985), Löhner *et al.* (1987, 1988), Baum and Löhner (1991), Sivier *et al.* (1992), Baum *et al.* (1993, 1995), Baum and Löhner (1994), Löhner *et al.* (1999)), detonation and flame propagation (Patnaik *et al.* (1987, 1989), magnetohydrodynamics (Scannapieco and Ossakow (1976)) and plasma physics. Tests have shown that FCT schemes compare very well, or in some cases are superior, to standard TVD schemes (Zalesak and Löhner (1989), Luo *et al.* (1993)) or even spectral schemes (deFainchtein *et al.* (1995)). This is surprising, considering that no detailed wave analysis is invoked during the limiting process.

10 EDGE-BASED COMPRESSIBLE FLOW SOLVERS

Consider the typical formation of a RHS using a finite element approximation with shape functions N^i. The resulting integrals to be evaluated are given by

$$\mathbf{r}^i = \int N^i \mathbf{r}(\mathbf{u})\, d\Omega = \sum_{el} \int N^i \mathbf{r}(N^j \mathbf{u}_j)\, d\Omega_{el}. \tag{10.1}$$

These integrals operate on two sets of data:

(a) point data, for $\mathbf{r}^i$, $\mathbf{u}^i$; and

(b) element data, for volumes, shape functions, etc.

The flow of information is as follows:

1. `GATHER` point information into the element (e.g. $\mathbf{u}_i$);
2. operate on element data to evaluate the integral in (10.1); and
3. `SCATTER-ADD` element RHS data to point data to obtain $\mathbf{r}^i$.

For many simple flow solvers the effort in step 2 may be minor compared to the cost of indirect addressing operations in steps 1 and 3. A way to reduce the indirect addressing overhead for low-order elements is to change the element-based data structure to an edge-based data structure. This eliminates certain redundancies of information in the element-based data structure. To see this more clearly, consider the formation of the RHS for the Laplacian operator on a typical triangulation. Equation (10.1) may be recast as

$$\mathbf{r}^i = K^{ij} u_j = \sum_{el} \mathbf{K}_{el} \mathbf{u}_{el} = \sum_{el} \mathbf{r}_{el}. \tag{10.1a}$$

This immediately opens three possibilities:

- obtain first the global matrix K^{ij} and store it in some optimal way (using so-called sparse storage techniques);
- perform a loop over elements, obtaining $\mathbf{r}_{el}$ and adding to $\mathbf{r}$; and
- obtain the off-diagonal coefficients K^{ij}, $i \neq j$, perform a loop over edges $i,\ j$, obtaining $\mathbf{r}_{ij}$ and adding to $\mathbf{r}$.

The most economic choice will depend on the number of times the RHS needs to be evaluated, how often the mesh changes, available storage, etc. The indirect addressing overhead incurred when performing a RHS evaluation for the Laplacian using linear elements, as well as the associated FLOPS, are given in Table 10.1. Recall that for a typical mesh of triangles `nelem=2*npoin`, `nedge=3*npoin`, `npsup=6*npoin`, whereas for a typical mesh of tetrahedra `nelem=5.5*npoin`, `nedge=7*npoin`, `npsup=14*npoin`.

Table 10.1. (a) Laplacian: Gather/Scatter overhead for linear elements. (b) FLOPS overhead: Laplacian/linear elements

	Element-Based	Edge-Based	Sparse
(a)			
2-D	3*3*nelem=18*npoin	3*2*nedge=18*npoin	6*npoin
3-D	3*4*nelem=66*npoin	3*2*nedge=42*npoin	14*npoin
(b)			
2-D	3*6*nelem= 36*npoin	4*nedge=12*npoin	12*npoin
3-D	4*8*nelem=176*npoin	4*nedge=28*npoin	28*npoin

The indirect addressing overhead incurred when performing a RHS evaluation for the Galerkin approximation to the advective fluxes and the associated FLOPS are given in Table 10.2.

Table 10.2. (a) Gather/Scatter overhead for linear elements. (b) FLOPS overhead: Galerkin fluxes for linear elements

	Element-Based	Edge-Based
(a)		
2-D	2*3*nelem=12*npoin	2*2*nedge=12*npoin
3-D	2*4*nelem=44*npoin	2*2*nedge=28*npoin
(b)		
3-D	22*nelem=121*npoin	7*nedge=49*npoin

As one can see, a considerable reduction of indirect addressing operations and FLOPS has been achieved by going to an edge-based data structure. Given their importance in general-purpose CFD codes, the present chapter is devoted to edge-based solvers. Starting from the by now familiar element integrals obtained for Galerkin weighted residual approximations, the resulting edge-based expressions are derived. Thereafter, a brief taxonomy of possible solvers is given.

10.1. The Laplacian operator

The geometrical expressions required when going from an element-based data structure to an edge-based data structure will now be derived. For simplicity and clarity we start with the Laplacian operator. The RHS in the domain is given by

$$r^i = -\int \nabla N^i \cdot \nabla N^j \, d\Omega \, \hat{u}_j = -\left[\sum_{el} \int \nabla N^i \cdot \nabla N^j \, d\Omega\right]\hat{u}_j. \qquad (10.2)$$

This integral can be split into those shape functions that are the same as N^i and those that are different ($j \neq i$):

$$r^i = -\sum_{j\neq i}\left[\sum_{el}\int \nabla N^i \cdot \nabla N^j \, d\Omega\right]\hat{u}_j - \sum_{el}\int \nabla N^i \cdot \nabla N^i \, d\Omega\, \hat{u}_i. \tag{10.3}$$

The conservation property of the shape functions

$$N^i_{,k} = -\sum_{j\neq i} N^j_{,k}, \tag{10.4}$$

allows (10.3) to be re-written as

$$r^i = -\sum_{j\neq i}\left[\sum_{el}\int \nabla N^i \cdot \nabla N^j \, d\Omega\right]\hat{u}_j + \left[\sum_{el}\int \nabla N^i \cdot \sum_{j\neq i} \nabla N^j \, d\Omega\right]\hat{u}_i, \tag{10.5}$$

or, after interchange of the double sums,

$$r^i = k^{ij}(\hat{u}_i - \hat{u}_j), \quad k^{ij} = \sum_{el}\int \nabla N^i \cdot \nabla N^j d\Omega, \quad j \neq i. \tag{10.6}$$

One may observe that:

- from a change in indices (*ij* versus *ji*) we obtain $k^{ji} = k^{ij}$; this is expected from the symmetry of the Laplacian operator;
- the edge and its sense of direction have to be defined; this is accomplished by
  - a *connectivity array* that stores the points of an edge
    `inpoed(1:2,nedge):= ip1, ip2`,
    and
  - taking `ip1 < ip2` to define the orientation;
- to build the stiffness coefficients k^{ij} we require the *edges of an element*, i.e. a list of the form `inedel(nedel,nelem):= ied1,ied2,....` The rapid construction of both `inpoed` and `inedel` have been described in Chapter 2.

A typical edge-based evaluation of the Laplacian RHS would look like the following:

```
rhspo(1:npoin)=0                                    ! Set points-RHS to zero
do iedge=1,nedge                                    ! Loop over the edges
  ipoi1=inpoed(1,iedge)                             ! 1st point of edge
  ipoi2=inpoed(2,iedge)                             ! 2nd point of edge
  ! Evaluate the edge-RHS
  redge=cedge(iedge)*(unkno(ipoi2)-unkno(ipoi1)
  rhspo(ipoi1)=rhspo(ipoi1)-redge                   ! Subtract from ipoi1
  rhspo(ipoi2)=rhspo(ipoi2)+redge                   ! Add to ipoi2
enddo
```

The conservation law expressed by the Laplacian as $\nabla \cdot \nabla u = 0$ is reflected on each edge: whatever is added to a point is subtracted from another. The flow of information and the sequence of operations performed in the loop are shown in Figure 10.1.

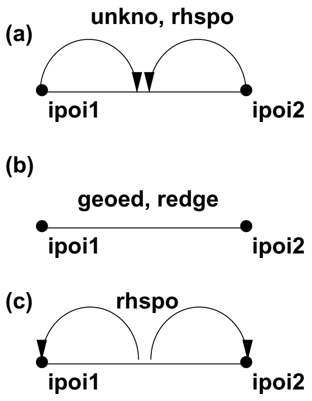


Figure 10.1. Edge-based Laplacian

10.2. First derivatives: first form

We now proceed to first derivatives. Typical examples are the Euler fluxes, the advective terms for pollution transport simulations or the Maxwell equations that govern electromagnetic wave propagation. The RHS is given by an expression of the form

$$\mathbf{r}^i = -\int N^i N^j_{,k} \, d\Omega \, \mathbf{F}^k_j, \tag{10.7}$$

where $\mathbf{F}^k_j$ denotes the flux in the kth dimension at node j. This integral is again separated into shape functions that are not equal to N^i and those that are equal:

$$\mathbf{r}^i = -\sum_{j\neq i}\bigg[\sum_{el}\int N^i N^j_{,k}\, d\Omega\bigg]\mathbf{F}^k_j - \sum_{el}\int N^i N^i_{,k}\, d\Omega\, \mathbf{F}^k_i. \tag{10.8}$$

As before, we use the conservation property (equation (10.4)) and get

$$\mathbf{r}^i = -\sum_{j\neq i}\bigg[\sum_{el}\int N^i N^j_{,k}\, d\Omega\bigg]\mathbf{F}^k_j + \bigg[\sum_{el}\int N^i \sum_{j\neq i} N^j_{,k}\, d\Omega\bigg]\mathbf{F}^k_i. \tag{10.9}$$

This may be restated as

$$\mathbf{r}^i = d^{ij}_k(\mathbf{F}^k_i - \mathbf{F}^k_j), \quad d^{ij}_k = \sum_{el} N^i N^j_{,k}\, d\Omega, \quad j\neq i. \tag{10.10}$$

One may observe that:

- for a change in indices *ij* versus *ji* we obtain

$$d^{ji}_k = -d^{ij}_k + \int_\Gamma N^j N^i n_k \, d\Gamma, \tag{10.11}$$

this is expected due to the unsymmetric operator;

- an extra boundary integral leads to a separate loop over boundary edges, adding (unsymmetrically) only to node j.

The flow of information for this first form of the first derivatives is shown in Figure 10.2.

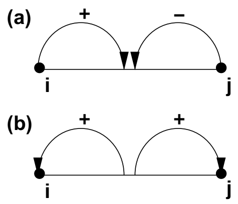


Figure 10.2. First derivatives: first form

Observe that we take a difference on the edge level, and then add contributions to both endpoints. This implies that the conservation law given for the first derivatives is not reflected at the edge level, although it is still maintained at the point level. This leads us to a second form, which reflects the conservation property on the edge level.

10.3. First derivatives: second form

While (10.10) is valid for any finite element shape function, a more desirable form of the RHS for the first-order fluxes is

$$\mathbf{r}^i = e_k^{ij}(\mathbf{F}_j^k + \mathbf{F}_i^k), \qquad e_k^{ij} = -e_k^{ji}. \tag{10.12}$$

In what follows, we will derive such an approximation for linear elements. As before, we start by separating the Galerkin integral into shape functions that are not equal to N^i and those that are equal:

$$\mathbf{r}^i = -\sum_{j\neq i}\left[\sum_{el}\int N^i N^j_{,k}\, d\Omega\right]\mathbf{F}_j^k - \sum_{el}\int N^i N^i_{,k}\, d\Omega\, \mathbf{F}_i^k. \tag{10.13}$$

In the following, we assume that whenever a sum over indices i, j is to be performed, then $i \neq j$, and that whenever the index i appears repeatedly in an expression, no sum is to be taken. Moreover, we use the abbreviation $\mathbf{F}_{ij}^k = \mathbf{F}_i^k + \mathbf{F}_j^k$. Integration by parts for the first integral yields

$$\mathbf{r}^i = \int_\Omega N^i_{,k} N^j\, d\Omega\, \mathbf{F}_j^k - \int_\Gamma N^i N^j n_k\, d\Gamma\, \mathbf{F}_j^k - \int_\Omega N^i N^i_{,k}\, d\Omega\, \mathbf{F}_i^k. \tag{10.14}$$

After conversion of the last domain integral into a surface integral via

$$\int_\Omega N^i N^i_{,k}\, d\Omega\, \mathbf{F}_i^k = \int_\Gamma N^i N^i n_k\, d\Gamma\, \mathbf{F}_i^k - \int_\Omega N^i N^i_{,k}\, d\Omega\, \mathbf{F}_i^k, \tag{10.15}$$

equation (10.13) may be recast as

$$\begin{aligned}\mathbf{r}^i = &\int_\Omega N^i_{,k} N^j\, d\Omega\, \mathbf{F}_{ij}^k - \int_\Omega N^i_{,k} N^j\, d\Omega\, \mathbf{F}_i^k - \int_\Gamma N^i N^j n_k\, d\Gamma\, \mathbf{F}_j^k \\ &- \frac{1}{2}\int_\Gamma N^i N^i n_k\, d\Gamma\, \mathbf{F}_i^k.\end{aligned} \tag{10.16}$$

By virtue of

$$\int_\Omega N^i_{,k} N^j \, d\Omega \, \mathbf{F}^k_{ij} = \int_\Gamma N^i N^j n_k \, d\Gamma \, \mathbf{F}^k_{ij} - \int_\Omega N^j_{,k} N^i \, d\Omega \, \mathbf{F}^k_{ij}, \qquad (10.17)$$

the first integral may be split to form

$$\begin{aligned}\mathbf{r}^i = {} & \frac{1}{2}\int_\Omega (N^i_{,k} N^j - N^j_{,k} N^i) \, d\Omega \, \mathbf{F}^k_{ij} + \frac{1}{2}\int_\Gamma N^i N^j n_k \, d\Gamma \, \mathbf{F}^k_{ij} \\ & - \int_\Omega N^i_{,k} N^j \, d\Omega \, \mathbf{F}^k_i - \int_\Gamma N^i N^j n_k \, d\Gamma \, \mathbf{F}^k_j - \frac{1}{2}\int_\Gamma N^i N^i n_k \, d\Gamma \, \mathbf{F}^k_i. \end{aligned} \qquad (10.18)$$

Given that for linear elements the shape-function derivatives are constant within each element, the last domain integral may be converted to

$$\int_\Omega N^i_{,k} N^j \, d\Omega \, \mathbf{F}^k_i = (nn-1)\int_\Omega N^i_{,k} N^i \, d\Omega \, \mathbf{F}^k_i = \frac{nn-1}{2}\int_\Gamma N^i N^i n_k \, d\Gamma \, \mathbf{F}^k_i, \qquad (10.19)$$

where *nn* denotes the number of nodes in the element, implying that

$$\begin{aligned}\mathbf{r}^i = {} & \frac{1}{2}\int_\Omega (N^i_{,k} N^j - N^j_{,k} N^i) \, d\Omega \, \mathbf{F}^k_{ij} - \frac{1}{2}\int_\Gamma N^i N^j n_k \, d\Gamma \, \mathbf{F}^k_{ij} \\ & + \int_\Gamma N^i N^j n_k \, d\Gamma \, \mathbf{F}^k_i - \frac{nn-2}{2}\int_\Gamma N^i N^i n_k \, d\Gamma \, \mathbf{F}^k_i. \end{aligned} \qquad (10.20)$$

The last two surface integrals only involve the values of fluxes at the same nodes as the residual is being evaluated. The interpolation property of shape functions, namely

$$\sum_m N^m(\mathbf{x}) = 1 \quad \forall \mathbf{x}, \qquad (10.21)$$

i.e.

$$\sum_{j \neq i} N^j = 1 - N^i, \qquad (10.22)$$

allows the $N^i N^j$ part of these integrals, which involves a sum over all possible $j \neq i$, to be re-written as

$$\int_\Gamma N^i N^j n_k \, d\Gamma \, \mathbf{F}^k_i = \int_\Gamma N^i (1 - N^i) n_k \, d\Gamma \, \mathbf{F}^k_i, \qquad (10.23)$$

leading to the final form of the residual

$$\begin{aligned}\mathbf{r}^i = {} & \frac{1}{2}\int_\Omega (N^i_{,k} N^j - N^j_{,k} N^i) \, d\Omega \, \mathbf{F}^k_{ij} - \frac{1}{2}\int_\Gamma N^i N^j n_k \, d\Gamma \, \mathbf{F}^k_{ij} \\ & + \int_\Gamma N^i \left(1 - \frac{nn}{2} N^i\right) n_k \, d\Gamma \, \mathbf{F}^k_i. \end{aligned} \qquad (10.24)$$

We therefore have an expression for the RHS of the form

$$\mathbf{r}^i = \mathbf{r}^i_{\Omega_{\text{edg}}} + \mathbf{r}^i_{\Gamma_{\text{edg}}} + \mathbf{r}^i_{\Gamma_{\text{pts}}}, \qquad (10.25)$$

where

$$\mathbf{r}^i_{\Omega_{\rm edg}} = \frac{1}{2}\int_\Omega (N^i_{,k}N^j - N^j_{,k}N^i)\, d\Omega\, (\mathbf{F}^k_j + \mathbf{F}^k_i) = d^{ij}_k(\mathbf{F}^k_i + \mathbf{F}^k_j), \tag{10.26a}$$

$$\mathbf{r}^i_{\Gamma_{\rm edg}} = -\frac{1}{2}\int_\Gamma N^i N^j n_k\, d\Gamma\, (\mathbf{F}^k_j + \mathbf{F}^k_i) = b^{ij}_k(\mathbf{F}^k_i + \mathbf{F}^k_j), \tag{10.26b}$$

$$\mathbf{r}^i_{\Gamma_{\rm pts}} = \int_\Gamma N^i\left(1 - \frac{nn}{2}N^i\right) n_k\, d\Gamma\, \mathbf{F}^k_i = b^i_k \mathbf{F}^k_i \tag{10.26c}$$

or

$$\mathbf{r}^i = d^{ij}_k(\mathbf{F}^k_j + \mathbf{F}^k_i) + b^{ij}_k(\mathbf{F}^k_j + \mathbf{F}^k_i) + b^i_k\mathbf{F}^k_i. \tag{10.27}$$

One may observe that this form is anti-symmetric in the pairing of indices i, j for d^{ij}_k and symmetric for b^{ij}_k. Moreover, an additive form for the fluxes on each edge has been achieved. However, it is important to bear in mind that this pairing of fluxes is *only possible for linear elements*.

A typical edge-based evaluation of the first derivatives RHS for a scalar variable would look like the following:

```
rhspo(1:npoin)=0                              ! Set points-RHS to zero
do iedge=1,nedge                              ! Loop over the edges
  ipoi1=inpoed(1,iedge)                       ! 1st point of edge
  ipoi2=inpoed(2,iedge)                       ! 2nd point of edge
  redge=cedge(1,iedge)*(fluxp(1,ipoi2)+fluxp(1,ipoi1)
&      +cedge(2,iedge)*(fluxp(2,ipoi2)+fluxp(2,ipoi1)
&      +cedge(3,iedge)*(fluxp(3,ipoi2)+fluxp(3,ipoi1)
  rhspo(ipoi1)=rhspo(ipoi1)-redge             ! Subtract from ipoi1
  rhspo(ipoi2)=rhspo(ipoi2)+redge             ! Add to ipoi2
enddo
```

The conservation law expressed by $\nabla \cdot \mathbf{F} = 0$ is reflected on each edge: whatever is added to a point is subtracted from another. The flow of information and the sequence of operations performed in the loop are shown in Figure 10.3.

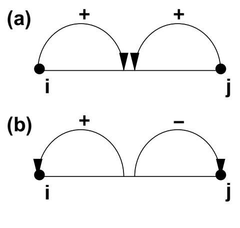


Figure 10.3. First derivatives: second form

10.4. Edge-based schemes for advection-dominated PDEs

The RHS obtained along edges was of the form

$$\mathbf{r}^i = d^{ij}_k(\mathbf{F}^k_i + \mathbf{F}^k_j). \tag{10.28}$$

The inner product over the dimensions k may be written in compact form as

$$\mathbf{r}^i = D^{ij}\mathcal{F}_{ij} = D^{ij}(\mathbf{f}_i + \mathbf{f}_j), \tag{10.29}$$

where the $\mathbf{f}_i$ are the 'fluxes along edges', obtained from the scalar product

$$\mathbf{f}_i = S_k^{ij}\mathbf{F}_i^k, \quad S_k^{ij} = \frac{d_k^{ij}}{D^{ij}}, \quad D^{ij} = \sqrt{d_k^{ij} d_k^{ij}} \tag{10.30}$$

and

$$d_k^{ij} = \frac{1}{2}\int_\Omega (N^i_{,k}N^j - N^j_{,k}N^i)\, d\Omega, \quad d_k^{ij} = -d_k^{ji}. \tag{10.31}$$

Observe that the switch from element-based flux evaluations to edge-based flux evaluations has serendipitously led to 'fluxes along edges'. This amounts to fluxes in one spatial dimension, opening the possibility to transplant the large body of theory developed over the last decades for 1-D finite difference and finite volume solvers (e.g. Sweby (1984), Toro (1999)) to unstructured grids in three dimensions. Along these lines, we note once more that for the standard Galerkin approximation we have

$$\mathcal{F}_{ij} = \mathbf{f}_i + \mathbf{f}_j. \tag{10.32}$$

Comparing this expression to a 1-D analysis, we see that it corresponds to a central difference approximation of the first derivative fluxes. This is known to be an unstable discretization, and must be augmented by stabilizing terms. In what follows, the most commonly used options are enumerated.

10.4.1. EXACT RIEMANN SOLVER (GODUNOV SCHEME)

The Galerkin approximation to the first derivatives resulted in (10.32), i.e. twice the average flux obtained from the unknowns $\mathbf{u}_i$, $\mathbf{u}_j$. If we assume that the flow variables are constant in the vicinity of the edge endpoints i, j, a discontinuity will occur at the edge midpoint. The evolution in time of this local flowfield was first obtained analytically by Riemann (1860), and consists of a shock, a contact discontinuity and an expansion wave. More importantly, the flux at the discontinuity remains constant in time. One can therefore replace the average flux of the Galerkin approximation by this so-called Riemann flux. This stable scheme, which uses the flux obtained from an exact Riemann solver, was first proposed by Godunov (1959). The flux is given by

$$\mathcal{F}_{ij} = 2f(\mathbf{u}_{ij}^R), \tag{10.33}$$

where $\mathbf{u}_{ij}^R$ is the local exact solution of the Riemann problem to the Euler equations, expressed as

$$\mathbf{u}_{lr}^R = Rie(\mathbf{u}_l, \mathbf{u}_r) \tag{10.34}$$

where

$$\mathbf{u}_r = \mathbf{u}_i, \quad \mathbf{u}_l = \mathbf{u}_j. \tag{10.35}$$

This is a first-order scheme. A scheme of higher-order accuracy can be achieved by a better approximation to $\mathbf{u}_r$ and $\mathbf{u}_l$, e.g., via a reconstruction process and monotone limiting (van Leer (1974), Colella (1990), Harten (1983), Woodward and Colella (1984), Barth (1991)). The major disadvantage of Godunov's approach is the extensive computational work introduced through the Riemann solver, as well as the limiting procedures. In the following, we will summarize the possible simplifications to this most expensive yet most 'accurate' of schemes, in order to arrive at schemes that offer better CPU versus accuracy ratios.

10.4.2. APPROXIMATE RIEMANN SOLVERS

A first simplification can be achieved by replacing the computationally costly exact Riemann solver by an approximate Riemann solver. A variety of possibilities have been considered (Roe (1981), Osher and Solomon (1982)). Among them, by far the most popular one is the one derived by Roe (1981). The first-order flux for this solver is of the form

$$\mathcal{F}_{ij} = \mathbf{f}_i + \mathbf{f}_j - |\mathbf{A}^{ij}|(\mathbf{u}_i - \mathbf{u}_j) \tag{10.36}$$

where $|\mathbf{A}^{ij}|$ denotes the standard Roe matrix evaluated in the direction $\mathbf{d}^{ij}$. In order to achieve a higher-order scheme, the amount of dissipation must be reduced. This implies reducing the magnitude of the difference $\mathbf{u}_i - \mathbf{u}_j$ by 'guessing' a smaller difference of the unknowns at the location where the approximate Riemann flux is evaluated (i.e. the middle of the edge). The assumption is made that the function behaves smoothly in the vicinity of the edge. This allows the construction or 'reconstruction' of alternate values for the unknowns at the middle of the edge, denoted by $\mathbf{u}_j^-$, $\mathbf{u}_i^+$, leading to a flux function of the form

$$\mathcal{F}_{ij} = \mathbf{f}^+ + \mathbf{f}^- - |A(\mathbf{u}_i^+, \mathbf{u}_j^-)|(\mathbf{u}_j^- - \mathbf{u}_i^+), \tag{10.37}$$

where

$$\mathbf{f}^+ = \mathbf{f}(\mathbf{u}_i^+), \quad \mathbf{f}^- = \mathbf{f}(\mathbf{u}_j^-). \tag{10.38}$$

The upwind-biased interpolations for $\mathbf{u}_i^+$ and $\mathbf{u}_j^-$ are defined by

$$\mathbf{u}_i^+ = \mathbf{u}_i + \tfrac{1}{4}[(1-k)\Delta_i^- + (1+k)(\mathbf{u}_j - \mathbf{u}_i)], \tag{10.39a}$$

$$\mathbf{u}_j^- = \mathbf{u}_j - \tfrac{1}{4}[(1-k)\Delta_j^+ + (1+k)(\mathbf{u}_j - \mathbf{u}_i)], \tag{10.39b}$$

where the forward and backward difference operators are given by

$$\Delta_i^- = \mathbf{u}_i - \mathbf{u}_{i-1} = 2\mathbf{l}_{ji} \cdot \nabla \mathbf{u}_i - (\mathbf{u}_j - \mathbf{u}_i), \tag{10.40a}$$

$$\Delta_j^+ = \mathbf{u}_{j+1} - \mathbf{u}_j = 2\mathbf{l}_{ji} \cdot \nabla \mathbf{u}_j - (\mathbf{u}_j - \mathbf{u}_i), \tag{10.40b}$$

and $\mathbf{l}_{ji}$ denotes the edge difference vector $\mathbf{l}_{ji} = \mathbf{x}_j - \mathbf{x}_i$. The parameter k can be chosen to control the degree of approximation (Hirsch (1991)). The additional information required for $\mathbf{u}_{i+1}$, $\mathbf{u}_{j+1}$ can be obtained in a variety of ways (see Figure 10.4):

- through continuation and interpolation from neighbouring elements (Billey *et al.* (1986));
- via extension along the most aligned edge (Weatherill *et al.* (1993b)); or
- by evaluation of gradients (Whitaker *et al.* (1989), Luo *et al.* (1993, 1994a)).

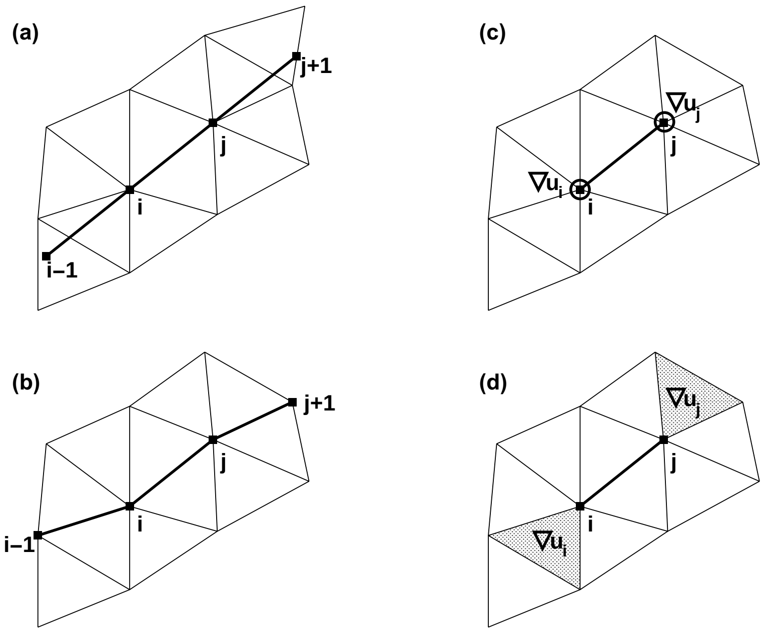


Figure 10.4. Higher-order approximations

The inescapable fact stated in Godunov's theorem that no linear scheme of order higher than one is free of oscillations implies that with these higher-order extensions, some form of limiting will be required. The flux limiter modifies the upwind-biased interpolations $\mathbf{u}_i$, $\mathbf{u}_j$, replacing them by

$$\mathbf{u}_i^+ = \mathbf{u}_i + \frac{s_i}{4}[(1 - ks_i)\Delta_i^- + (1 + ks_i)(\mathbf{u}_j - \mathbf{u}_i)], \tag{10.41a}$$

$$\mathbf{u}_j^- = \mathbf{u}_j - \frac{s_j}{4}[(1 - ks_j)\Delta_j^+ + (1 + ks_j)(\mathbf{u}_j - \mathbf{u}_i)], \tag{10.41b}$$

where s is the flux limiter. For $s = 0, 1$, the first- and high-order schemes are recovered, respectively. A number of limiters have been proposed in the literature, and this area is still a matter of active research (see Sweby (1984) for a review). We include the van Albada limiter here, one that is commonly used. This limiter acts in a continuously differentiable manner and is defined by

$$s_i = \max\left\{0, \frac{2\Delta_i^-(\mathbf{u}_j - \mathbf{u}_i) + \epsilon}{(\Delta_i^-)^2 + (\mathbf{u}_j - \mathbf{u}_i)^2 + \epsilon}\right\}, \tag{10.42a}$$

$$s_j = \max\left\{0, \frac{2\Delta_j^+(\mathbf{u}_j - \mathbf{u}_i) + \epsilon}{(\Delta_j^+)^2 + (\mathbf{u}_j - \mathbf{u}_i)^2 + \epsilon}\right\}, \tag{10.42b}$$

where ϵ is a very small number to prevent division by zero in smooth regions of the flow. For systems of PDEs a new question arises: which variables should one use to determine the limiter function s? Three (and possibly many more) possibilities can be considered:

conservative variables, primitive variables and characteristic variables. Using limiters on characteristic variables seems to give the best results, but due to the lengthy algebra this option is very costly. For this reason, primitive variables are more often used for practical calculations as they provide a better accuracy versus CPU ratio. Before going on, we remark that the bulk of the improvement when going from the first-order scheme given by (10.36) to higher-order schemes stems from the decrease in dissipation when $\mathbf{u}_i - \mathbf{u}_j$ is replaced by $\mathbf{u}_j^- - \mathbf{u}_i^+$. In most cases, the change from $\mathbf{f}_i + \mathbf{f}_j$ to $\mathbf{f}^+ + \mathbf{f}^-$ only has a minor effect and may be omitted.

10.4.3. SCALAR LIMITED DISSIPATION

A further possible simplification can be made by replacing the Roe matrix by its spectral radius. This leads to a numerical flux function of the form

$$\mathcal{F}_{ij} = \mathbf{f}_i + \mathbf{f}_j - |\lambda^{ij}|(\mathbf{u}_j - \mathbf{u}_i), \tag{10.43}$$

where

$$|\lambda^{ij}| = |v_{ij}^k \cdot S_k^{ij}| + c^{ij}, \tag{10.44}$$

and v_{ij}^k and c^{ij} denote edge values, computed as nodal averages, of the fluid velocity and speed of sound, respectively. This can be considered as a centred difference scheme plus a second-order dissipation operator, leading to a first-order, monotone scheme. As before, a higher-order scheme can be obtained by a better approximation to the 'right' and 'left' states of the 'Riemann problem', which have been set to $\mathbf{u}_r = \mathbf{u}_i$, $\mathbf{u}_l = \mathbf{u}_j$. This reduces the difference between $\mathbf{u}_i$, $\mathbf{u}_j$, decreasing in turn the dissipation. The resulting flux function is given by

$$\mathcal{F}_{ij} = \mathbf{f}_i + \mathbf{f}_j - |\lambda_{ij}|(\mathbf{u}_j^- - \mathbf{u}_i^+), \tag{10.45}$$

with $\mathbf{u}_j^-$, $\mathbf{u}_i^+$ given by (10.41) and (10.42). Consider the special case of smooth flows where the limiters are switched off, i.e. $s_i = s_j = 1$. This results in

$$\mathbf{u}_i^+ = \mathbf{u}_i + \tfrac{1}{4}[(1-k)[2\mathbf{l}_{ji} \cdot \nabla \mathbf{u}_i - (\mathbf{u}_j - \mathbf{u}_i)] + (1+k)(\mathbf{u}_j - \mathbf{u}_i)], \tag{10.46a}$$

$$\mathbf{u}_j^- = \mathbf{u}_j - \tfrac{1}{4}[(1-k)[2\mathbf{l}_{ji} \cdot \nabla \mathbf{u}_j - (\mathbf{u}_j - \mathbf{u}_i)] + (1+k)(\mathbf{u}_j - \mathbf{u}_i)], \tag{10.46b}$$

leading to a dissipation operator of the form

$$\mathbf{u}_j^- - \mathbf{u}_i^+ = (1-k)[\mathbf{u}_i - \mathbf{u}_j + \tfrac{1}{2}\mathbf{l}_{ji} \cdot (\nabla \mathbf{u}_i + \nabla \mathbf{u}_j)], \tag{10.47}$$

i.e. *fourth-order* dissipation. The important result is that schemes that claim not to require a fourth-order dissipation inherently do so through limiting. The fact that at least some form of fourth-order dissipation is required to stabilize hyperbolic systems of PDEs is well known in the mathematical literature.

10.4.4. SCALAR DISSIPATION WITH PRESSURE SENSORS

Given that for smooth problems the second-order dissipation $|\mathbf{u}_i - \mathbf{u}_j|$ reverts to a fourth order, and that limiting requires a considerable number of operations, the next possible simplification is to replace the limiting procedure by a pressure sensor function. A scheme of this type may be written as

$$\mathcal{F}_{ij} = \mathbf{f}_i + \mathbf{f}_j - |\lambda_{ij}|\left[\mathbf{u}_i - \mathbf{u}_j + \frac{\beta}{2}\mathbf{l}_{ji} \cdot (\nabla\mathbf{u}_i + \nabla\mathbf{u}_j)\right], \tag{10.48}$$

where $0 < \beta < 1$ denotes a pressure sensor function of the form (Peraire *et al.* (1992a))

$$\beta = 1 - \frac{p_i - p_j - 0.5\,\mathbf{l}_{ji} \cdot (\nabla p_i + \nabla p_j)}{|p_i - p_j| + |0.5\,\mathbf{l}_{ji} \cdot (\nabla p_i + \nabla p_j)|}. \tag{10.49}$$

For $\beta = 0, 1$, second- and fourth-order damping operators are obtained. Several forms are possible for the sensor function β (Mestreau *et al.* (1993)). The following two-pass strategy is one that has proven reliable. In a first pass over the mesh, the highest of the edge-based β-values given by (10.49) is kept for each point. In a second pass the highest value of the two points belonging to an edge is kept as the final β-value. Although this discretization of the Euler fluxes looks like a blend of second- and fourth-order dissipation, it has no adjustable parameters.

10.4.5. SCALAR DISSIPATION WITHOUT GRADIENTS

The scalar dissipation operator presented above still requires the evaluation of gradients. This can be quite costly for Euler simulations: for a typical multistage scheme, more than 40% of the CPU time is spent in gradient operations, even if a new dissipation operator is only required at every other stage. The reason for this lies in the very large number of gradients required: 15 for the unknowns in three dimensions, and an additional three for the pressure. An alternative would be to simplify the combination of second- and fourth-order damping operators by writing these operators out explicitly:

$$d_2 = \lambda_{ij}(1-\beta)[\mathbf{u}_i - \mathbf{u}_j], \quad d_4 = \lambda_{ij}\beta\left[\mathbf{u}_i - \mathbf{u}_j + \frac{\mathbf{l}_{ji}}{2} \cdot (\nabla\mathbf{u}_i + \nabla\mathbf{u}_j)\right]. \tag{10.50}$$

Performing a Taylor expansion in the direction of the edge, we have

$$\mathbf{u}_i - \mathbf{u}_j + \frac{\mathbf{l}_{ji}}{2} \cdot (\nabla\mathbf{u}_i + \nabla\mathbf{u}_j) \approx \frac{\mathbf{l}_{ji}^2}{4}\left[\left.\frac{\partial^2\mathbf{u}}{\partial l^2}\right|_j - \left.\frac{\partial^2\mathbf{u}}{\partial l^2}\right|_i\right]. \tag{10.51}$$

This suggests the following simplification, which neglects the off-diagonal terms of the tensor of second derivatives,

$$\frac{l^2}{4}\left[\left.\frac{\partial^2\mathbf{u}}{\partial l^2}\right|_j - \left.\frac{\partial^2\mathbf{u}}{\partial l^2}\right|_i\right] \approx \frac{l^2}{4}[\nabla^2\mathbf{u}_j - \nabla^2\mathbf{u}_i], \tag{10.52}$$

and leads to the familiar blend of second- and fourth-order damping operators (Jameson *et al.* (1981), Mavriplis (1991b))

$$\mathcal{F}_{ij} = \mathbf{f}_i + \mathbf{f}_j - |\lambda_{ij}|(1-\beta)[\mathbf{u}_i - \mathbf{u}_j] - |\lambda_{ij}|\beta\frac{l^2}{4}[\nabla^2\mathbf{u}_j - \nabla^2\mathbf{u}_i]. \tag{10.53}$$

10.4.6. TAYLOR–GALERKIN SCHEMES

Owing to their importance for transient calculations, it is worth considering possible edge-based Taylor–Galerkin schemes. The essential feature of any Taylor–Galerkin scheme is the combination of time and space discretizations, leading to second-order accuracy in both time and space. An edge-based two-step Taylor–Galerkin scheme can readily be obtained by setting the numerical flux to

$$\mathcal{F}_{ij} = 2\mathbf{f}(\mathbf{u}_{ij}^{n+\frac{1}{2}}), \tag{10.54}$$

where

$$\mathbf{u}_{ij}^{n+\frac{1}{2}} = \frac{1}{2}(\mathbf{u}_i + \mathbf{u}_j) - \frac{\Delta t}{2}\frac{\partial \mathbf{f}^j}{\partial x_j}\bigg|_{ij}, \tag{10.55}$$

and $(\partial \mathbf{f}^j/\partial x_j)|_{ij}$ is computed on each edge and given by either

$$\frac{\partial \mathbf{f}^j}{\partial x_j}\bigg|_{ij} = \frac{\mathbf{l}_{ji}}{\mathbf{l}_{ji}^2}\cdot(\mathbf{F}^i - \mathbf{F}^j) \tag{10.56}$$

or

$$\frac{\partial \mathbf{f}^j}{\partial x_j}\bigg|_{ij} = \frac{\mathbf{f}_i - \mathbf{f}_j}{D^{ij}}. \tag{10.57}$$

The major advantage of this scheme lies in its speed, since there is no requirement of gradient computations, as well as limiting procedures for smooth flows. An explicit numerical dissipation (e.g. in the form of a Lapidus viscosity) is needed to model flows with discontinuities. Taylor–Galerkin schemes by themselves are of little practical use for problems with strong shocks or other discontinuities. However, they provide useful high-order schemes for the flux-corrected transport schemes presented below.

10.4.7. FLUX-CORRECTED TRANSPORT SCHEMES

The idea behind FCT (Boris and Book (1973, 1976), Book and Boris (1975), Zalesak (1979)) is to combine a high-order scheme with a low-order scheme in such a way that the high-order scheme is employed in smooth regions of the flow, whereas the low-order scheme is used near discontinuities in a conservative way, in an attempt to yield a monotonic solution. The implementation of an edge-based FCT scheme is exactly the same as its element-based counterpart (Löhner *et al.* (1987, 1998)). The use of edge-based data structures makes the implementation more efficient, and this is especially attractive for 3-D problems. The edge-based two-step Taylor–Galerkin scheme will lead to a high-order increment of the form

$$\mathbf{M}_l\Delta\mathbf{u}^h = \mathbf{r} + (\mathbf{M}_l - \mathbf{M}_c)\Delta\mathbf{u}^h. \tag{10.58}$$

Here $\mathbf{M}_l$ denotes the diagonal, lumped mass matrix and $\mathbf{M}_c$ the consistent finite element mass matrix. The low-order scheme is simply

$$\mathbf{M}_l\Delta\mathbf{u}^l = \mathbf{r} + c_d(\mathbf{M}_c - \mathbf{M}_l)\mathbf{u}^n, \tag{10.59}$$

i.e. lumped mass matrix plus a lot of diffusion. Subtracting (10.59) from (10.58) yields the antidiffusive edge contributions

$$(\Delta\mathbf{u}^h - \Delta\mathbf{u}^l) = \mathbf{M}_l^{-1}(\mathbf{M}_l - \mathbf{M}_c)(c_d\mathbf{u}^n + \Delta\mathbf{u}^h). \tag{10.60}$$

This avoids any need for physical flux recomputations, leading to a very fast overall scheme. As with other limiters (see above), for systems of PDEs one faces several possible choices for the variables on which to limit.

Table 10.3 summarizes the main ingredients of high-resolution schemes for compressible flows, indirectly comparing the cost of most current flow solvers.

Table 10.3. Ingredients of edge-based compressible flow solvers

Solver	Riemann	Gradient	Char. Transf.	Limiting
Classic Godunov	Yes	Yes	Yes	Yes
Consvar Godunov	Yes	Yes	No	Yes
Consvar Roe/HLLC	Approx	Yes	No	Yes
Central/2/lim	No	Yes	No	Yes
Central/2/4	No	Yes	No	No
Central/2/4/Laplacian	No	No	No	No
Central/2	No	No	No	No
Central	No	No	No	No
Taylor–Galerkin	No	No	No	No
Taylor–Galerkin FCT	No	No	No	Yes

‘Char. Transf.’ denotes transformation to characteristic variable, and ‘Roe’, ‘HLLC’ denote approximate Riemann solvers.

11 INCOMPRESSIBLE FLOW SOLVERS

Among the flows that are of importance and interest to mankind, the category of low Mach-number or incompressible flows is by far the largest. Most of the manufactured products we use on a daily basis will start their life as an incompressible flow (polymer extrusion, melts, a large number of food products, etc.). The air which surrounds us can be considered, in almost all instances, as an incompressible fluid (airplanes flying at low Mach numbers, flows in and around cars, vans, buses, trains and buildings). The same applies to water (ships, submarines, torpedoes, pipes, etc.) and most biomedical liquids (e.g. blood). Given this large number of possible applications, it is not surprising that numerical methods to simulate incompressible flows have been developed for many years, as evidenced by an abundance of literature (various conferences[1], Thomasset (1981), Gunzburger and Nicolaides (1993), Hafez (2003)).

The equations describing incompressible, Newtonian flows may be written as

$$\mathbf{v}_{,t} + \mathbf{v}\nabla\mathbf{v} + \nabla p = \nabla\mu\nabla\mathbf{v}, \tag{11.1}$$

$$\nabla \cdot \mathbf{v} = 0. \tag{11.2}$$

Here p denotes the pressure, $\mathbf{v}$ the velocity vector and both the pressure p and the viscosity μ have been normalized by the (constant) density ρ. By taking the divergence of (11.1) and using (11.2) we can immediately derive the so-called pressure-Poisson equation

$$\nabla^2 p = -\nabla \cdot \mathbf{v}\nabla\mathbf{v}. \tag{11.3}$$

What sets incompressible flow solvers apart from compressible flow solvers is the fact that the pressure is not obtained from an equation of state $p = p(\rho, T)$, but from the divergence constraint. This implies that the pressure field establishes itself instantaneously (reflecting the infinite speed of sound assumption of incompressible fluids) and must therefore be integrated implicitly in time. From a numerical point of view, the difficulties in solving (11.1)–(11.3) are the usual ones. First-order derivatives are problematic, while second-order derivatives can be discretized by a straightforward Galerkin approximation. We will first treat the advection operator and then proceed to the divergence operator.

11.1. The advection operator

As with the compressible Euler/Navier–Stokes equations, there are three ways of modifying the unstable Galerkin discretization of the advection terms:

[1] See the following conference series: *Finite Elements in Fluids* I–IX, John Wiley & Sons; *Int. Conf. Num. Meth. Fluid Dyn.* I–XII, Springer Lecture Notes in Physics; *AIAA CFD Conf.* I–XII, AIAA CP; *Num. Meth. Laminar and Turbulent Flow*, Pineridge Press, and others.

(a) integration along characteristics;

(b) Taylor–Galerkin (or streamline diffusion); and

(c) edge-based upwinding.

11.1.1. INTEGRATION ALONG CHARACTERISTICS

If we consider the advection of a scalar quantity ϕ, the Eulerian description

$$\phi_{,t} + \mathbf{v} \cdot \nabla\phi = 0 \tag{11.4}$$

can be recast in the Lagrangian frame

$$\phi_{,t} = 0, \tag{11.5}$$

i.e. there should be no change in the unknown along the characteristics given by the instantaneous streamlines. This implies that if we desire to know the value of $\phi_i^{n+1} = \phi_i(t + \Delta t)$ at a given point $\mathbf{x}_i$, we can integrate in the upstream direction and locate the position $\mathbf{x}'_i$ where the particle was at time $t - \Delta t$. By virtue of (11.5) we then have

$$\phi_i(\mathbf{x}_i, t + \Delta t) = \phi(\mathbf{x}'_i, t). \tag{11.6}$$

The integration along the streamlines is usually carried out using multistage Runge–Kutta schemes (Gregoire *et al.* (1985)). Employing linear interpolation for (11.6) results in a monotonic, first-order scheme. Higher-order interpolation has been attempted, but can result in overshoots and should be used in conjunction with limiters.

11.1.2. TAYLOR–GALERKIN

In order to derive this type of approximation, we start with the following Taylor expansion:

$$\Delta\mathbf{v} = \Delta t \mathbf{v}_{,t}|^n + \frac{\Delta t^2}{2}\mathbf{v}_{,tt}\bigg|^{n+\theta}. \tag{11.7}$$

Inserting (11.1) repeatedly into (11.4), using (11.2), and ignoring any spatial derivatives of order higher than two yields

$$\begin{aligned}\Delta\mathbf{v} = {} & \Delta t[-\mathbf{v}\cdot\nabla\mathbf{v} - \nabla p + \nabla\mu\nabla\mathbf{v}] + \theta\Delta t[-\nabla\Delta p + \nabla\mu\nabla\Delta\mathbf{v}] \\ & + \frac{\Delta t^2}{2}[\nabla\mathbf{v}\otimes\mathbf{v}\nabla\mathbf{v} + (\mathbf{v}\cdot\nabla\mathbf{v} + \nabla p)\cdot\nabla\mathbf{v} + \mathbf{v}\cdot\nabla\nabla p]^{n+\theta}.\end{aligned} \tag{11.8}$$

Observe the following.

(a) An additional factor appears in front of the streamline upwind diffusion (Brooks and Hughes (1982)) or balancing tensor diffusivity (Kelly *et al.* (1980)). Usually, only the advection-diffusion equation is studied. This assumes that the transport velocity field is steady. In the present case, the velocity field itself is being convected. This introduces the additional factor.

(b) Additional ‘mixed’ velocity–pressure terms appear as a result of the consistent treatment of time advancement for all terms of the Navier–Stokes equations. They may be interpreted as upwind factors for the pressure.

(c) The additional factors depend on Δt^2. This implies that, for steady flow problems, the results will depend on the timestep chosen. This shortcoming is often avoided by replacing Δt^2 with $\Delta t\Delta\tau$, where the upwinding factor $\Delta\tau$ is dependent on the local mesh size and the unknowns. The large number of so-called Petrov–Galerkin and Galerkin least-squares schemes that have been developed over the past decade all fall under this category.

11.1.3. EDGE-BASED UPWINDING

The third option to treat the advection terms is to derive a consistent numerical flux for edge-based solvers. The Galerkin approximation for the advection terms yields a RHS of the form

$$r^i = D^{ij}\mathcal{F}_{ij} = D^{ij}(\mathbf{f}_i + \mathbf{f}_j), \tag{11.9}$$

where the $\mathbf{f}_i$ are the ‘fluxes along edges’,

$$\mathbf{f}_i = S_k^{ij}\mathbf{F}_i^k, \quad S_k^{ij} = \frac{d_k^{ij}}{D^{ij}}, \quad D^{ij} = \sqrt{d_k^{ij}d_k^{ij}}, \tag{11.10}$$

$$\mathcal{F}_{ij} = \mathbf{f}_i + \mathbf{f}_j, \tag{11.11}$$

$$\mathbf{f}_i = (S_k^{ij}v_i^k)\mathbf{v}_i, \quad \mathbf{f}_j = (S_k^{ij}v_j^k)\mathbf{v}_j. \tag{11.12}$$

A consistent numerical flux is given by

$$\mathcal{F}_{ij} = \mathbf{f}_i + \mathbf{f}_j - |v^{ij}|(\mathbf{v}_i - \mathbf{v}_j), \tag{11.13}$$

where

$$v^{ij} = \tfrac{1}{2}S_k^{ij}(v_i^k + v_j^k). \tag{11.14}$$

As before, this first-order scheme can be improved by reducing the difference $\mathbf{v}_i - \mathbf{v}_j$ through (limited) extrapolation to the edge centre (Löhner *et al.* (1999)).

11.2. The divergence operator

A persistent difficulty with incompressible flow solvers has been the derivation of a stable scheme for the divergence constraint (11.2). The stability criterion for the divergence constraint is also known as the Ladyzenskaya–Babuska–Brezzi or LBB condition (Gunzburger (1987)). To identify the potential difficulties, let us consider the pseudo-Stokes problem:

$$\mathbf{v} + \nabla p = 0, \tag{11.15}$$

$$\nabla \cdot \mathbf{v} = 0, \tag{11.16}$$

which also satisfies

$$\nabla^2 p = 0. \tag{11.17}$$

A Galerkin or central-difference discretization will result in a matrix system of the form

$$\begin{Bmatrix} \mathbf{M} & \mathbf{G} \\ \mathbf{D} & \mathbf{0} \end{Bmatrix} \cdot \begin{pmatrix} \mathbf{v} \\ \mathbf{p} \end{pmatrix} = \begin{pmatrix} \mathbf{r}_v \\ \mathbf{r}_p \end{pmatrix}. \tag{11.18}$$

Solving for **p**, we obtain

$$\mathbf{DM}^{-1}\mathbf{Gp} = \mathbf{DM}^{-1}\mathbf{r}_v - \mathbf{r}_p, \tag{11.19}$$

implying

$$\nabla^2 p \approx \mathbf{DM}^{-1}\mathbf{Gp}. \tag{11.20}$$

The approximate Laplacian operator obtained for the pressure from the solution of the matrix problem (11.15) may be unstable. This can be readily seen for a 1-D grid with uniform element size h. The resulting operators are listed below in Table 11.1.

Table 11.1. $\mathbf{DM}^{-1}\mathbf{G}$ for different velocity/pressure combinations

Velocity	Pressure	Operator
p1	q0	$(\ \ 0, -1, 2, -1, \ \ 0)$
p1	p1	$(-1, \ \ 0, 2, \ \ 0, -1)$
iso-p1	p1	$(-1, -1, 4, -1, -1)$

In particular, we note that the p1/p1 pseudo-Laplacian, which is equivalent to the stencil obtained from repeated first-order derivatives, is a five-point stencil that admits the unstable chequerboard mode shown in Figure 11.1 as one of its solutions. The p1/iso-p1 element is a mix of the p1/p1 five-point stencil and the usual three-point stencil. This discretization is stable.

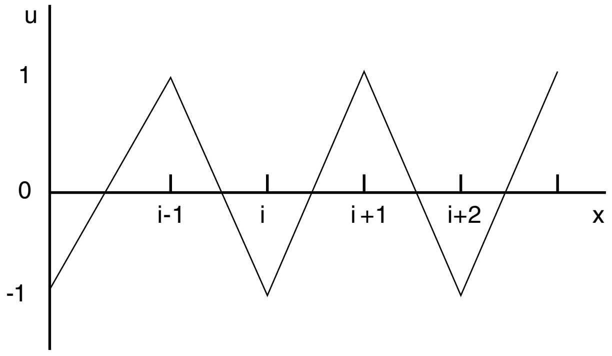


Figure 11.1. Chequerboard mode on a uniform 1-D grid

The classic way to satisfy the LBB condition has been to use different functional spaces for the velocity and pressure discretization (Fortin and Thomasset (1979)). Typically, the velocity space has to be richer, containing more degrees of freedom than the pressure space. Elements belonging to this class are the p1/p1+bubble mini-element (Soulaimani *et al.* (1987)), the p1/iso-p1 element (Thomasset (1981)) and the p1/p2 element

(Taylor and Hood (1973)). An alternative way to satisfy the LBB condition is through the use of artificial viscosities (Löhner (1990a)), 'stabilization' (Franca *et al.* (1989), Tezduyar (1992), Franca and Frey (1992)) (a more elegant term for the same thing) or a consistent numerical flux (yet another elegant term). To see the equivalency of these approaches, we re-state the divergence constraint as

$$c^{-2}p_{,t} + \nabla \cdot \mathbf{v} = 0, \tag{11.21}$$

where c is the speed of sound, and then proceed with the following Taylor expansion:

$$c^{-2}\Delta p = c^{-2}\Delta t\, p_{,t}|^n + c^{-2}\frac{\Delta t^2}{2}p_{,tt}\bigg|^{n+\theta}. \tag{11.22}$$

Inserting (11.18) repeatedly into (11.19), using (11.1), ignoring any spatial derivatives of order higher than two and taking the limit $c \to \infty$ yields

$$\Delta t\nabla \cdot \mathbf{v} = \frac{\Delta t^2}{2}[\nabla^2 p + \nabla \cdot \mathbf{v} \cdot \nabla \mathbf{v}]^{n+\theta}. \tag{11.23}$$

Observe the following.

- A Laplacian 'pressure-diffusion' appears naturally on the RHS. We will return to this Laplacian subsequently when we look for appropriate spatial discretizations.
- An additional 'mixed' velocity–pressure term appears as a result of the consistent treatment in the time advancement for the divergence equation.

Thus, given the divergence constraint, two alternate routes may be taken:

(a) spend more effort per element with the original (simpler) equations by using different shape-function spaces; or

(b) spend more effort per equation with a simpler element.

The complete equivalency of both approaches has been repeatedly demonstrated (e.g. Soulaimani *et al.* (1987), Löhner (1990a)). Production codes place a requirement for simplicity. A complex element, however good, will always be less favoured by the user as well as by those maintaining and upgrading codes. It is for this reason that simpler elements with a modified or augmented divergence constraint are commonly used in large-scale CFD codes.

The second option to treat the divergence constraint is to use consistent edge-based numerical fluxes. The Galerkin fluxes are given by

$$\mathcal{F}_{ij} = \mathbf{f}_i + \mathbf{f}_j, \quad \mathbf{f}_i = S_k^{ij} v_i^k, \quad \mathbf{f}_j = S_k^{ij} v_j^k. \tag{11.24}$$

A consistent numerical flux may be constructed by adding pressure terms of the form

$$\mathcal{F}_{ij} = \mathbf{f}_i + \mathbf{f}_j - |\lambda^{ij}|(p_i - p_j) \tag{11.25}$$

where the eigenvalue λ^{ij} is given by the ratio of the characteristic advective timestep of the edge Δt and the characteristic advective length of the edge l:

$$\lambda^{ij} = \frac{\Delta t^{ij}}{l^{ij}}. \tag{11.26}$$

Higher-order schemes can be derived by reconstruction and limiting, or by substituting the first-order differences of the pressure with third-order differences:

$$\mathcal{F}_{ij} = \mathbf{f}_i + \mathbf{f}_j - |\lambda^{ij}|\left(p_i - p_j + \frac{l^{ij}}{2}(\nabla p_i + \nabla p_j)\right). \tag{11.27}$$

This results in a stable, low-diffusion, fourth-order damping for the divergence constraint.

11.3. Artificial compressibility

A very elegant way to circumvent the problems associated with the advection and divergence operators is to recast the incompressible Euler/Navier–Stokes equations into a hyperbolic/parabolic system by adding the time derivative of the pressure to the divergence constraint (Chorin (1967)),

$$\frac{1}{c^2}p_{,t} + \nabla \cdot \mathbf{v} = 0. \tag{11.28}$$

The 'velocity of sound' of the hyperbolic system is given by c and the eigenvalues are $(u + c, u, u, u, u - c)$. Once the system has been written in this form, all the schemes developed for the compressible Euler/Navier–Stokes equations can be used. In particular, simple fourth-order artificial viscosity expressions of the form

$$\lambda_{\max} h^3(\nabla^4\mathbf{v}, \nabla^4 p) \tag{11.29}$$

can readily be used (Rizzi and Eriksson (1985), Hino *et al.* (1993), Farmer *et al.* (1993), Peraire *et al.* (1994), Kallinderis and Chen (1996), Hino (1997)). Alternatively, Roe schemes or more elaborate approximate 'Riemann solvers' can be employed. Note the similarity between (11.29) and (11.27). From an artificial compressibility perspective, the addition of stabilizing terms based on the pressure p for the divergence constraint is natural.

11.4. Temporal discretization: projection schemes

The hyperbolic character of the advection operator and the elliptic character of the pressure-Poisson equation have led to a number of so-called projection schemes. The key idea is to predict first a velocity field from the current flow variables without taking the divergence constraint into account. In a second step, the divergence constraint is enforced by solving a pressure-Poisson equation. The velocity increment can therefore be separated into an advective and pressure increment:

$$\mathbf{v}^{n+1} = \mathbf{v}^n + \Delta\mathbf{v}^a + \Delta\mathbf{v}^p = \mathbf{v}^{**} + \Delta\mathbf{v}^p. \tag{11.30}$$

For an explicit integration of the advective terms, one complete timestep is given by:

(a) *advective/diffusive prediction:* $\mathbf{v}^n \to \mathbf{v}^{**}$

$$\left[\frac{1}{\Delta t} - \nabla\mu\nabla\right] \cdot (\mathbf{v}^{**} - \mathbf{v}^n) + \mathbf{v}^n \cdot \nabla\mathbf{v}^n = \nabla\mu\nabla\mathbf{v}^n; \tag{11.31a}$$

(b) *pressure correction:* $p^n \rightarrow p^{n+1}$

$$\nabla \cdot \mathbf{v}^{n+1} = 0;$$
$$\mathbf{v}^{n+1} + \Delta t \nabla p^{n+1} = \mathbf{v}^{**};$$

$\Rightarrow$

$$\nabla^2 p^{n+1} = \frac{\nabla \cdot \mathbf{v}^{**}}{\Delta t}; \tag{11.31b}$$

(c) *velocity correction*: $\mathbf{v}^{**} \rightarrow \mathbf{v}^{n+1}$

$$\mathbf{v}^{n+1} = \mathbf{v}^{**} - \Delta t \nabla p^{n+1}. \tag{11.31c}$$

This scheme was originally proposed by Chorin (1968), and has since been used repeatedly within finite difference (Kim and Moin (1985)), finite volume (Hino (1989), finite element (Gresho *et al.* (1982), Donea *et al.* (1982), Huffenus and Khaletzky (1984), Gresho and Chan (1990), Jue (1991)) and spectral element (Patera (1984)) solvers. The main drawback of this scheme is that as the contributions of pressure gradients are neglected in (11.31a) the residuals of the pressure correction do not vanish at a steady state, implying that the results depend on the timestep Δt. This situation can be remedied by considering the pressure for the advective/diffusive predictor. The resulting scheme is given by:

(a) *advective-diffusive prediction*: $\mathbf{v}^n \rightarrow \mathbf{v}^*$

$$\left[\frac{1}{\Delta t} - \nabla \mu \nabla\right](\mathbf{v}^* - \mathbf{v}^n) + \mathbf{v}^n \cdot \nabla \mathbf{v}^n + \nabla p^n = \nabla \mu \nabla \mathbf{v}; \tag{11.32a}$$

(b) *pressure correction*: $p^n \rightarrow p^{n+1}$

$$\nabla \cdot \mathbf{v}^{n+1} = 0;$$
$$\frac{\mathbf{v}^{n+1} - \mathbf{v}^*}{\Delta t} + \nabla(p^{n+1} - p^n) = 0;$$

which results in

$$\nabla^2(p^{n+1} - p^n) = \frac{\nabla \cdot \mathbf{v}^*}{\Delta t}; \tag{11.32b}$$

(c) *velocity correction*: $\mathbf{v}^* \rightarrow \mathbf{v}^{n+1}$

$$\mathbf{v}^{n+1} = \mathbf{v}^* - \Delta t \nabla(p^{n+1} - p^n). \tag{11.32c}$$

At a steady state, the residuals of the pressure correction vanish, implying that the result does not depend on the timestep Δt. Another advantage of this scheme as compared to the one given by (11.31a–c) is that the 'pressure-Poisson' equation (11.32b) computes *increments* of pressures, implying that the Dirichlet and Neumann boundary conditions simplify. Even for cases with gravity, $(\Delta p)_{,n} = 0$, whereas $p_{,n} = \rho \mathbf{g}$, where **g** denotes the gravity vector. From an implementational point of view, more cases can be handled with the default boundary conditions $\Delta p = 0$, $(\Delta p)_{,n} = 0$ for (11.32b) as compared to (11.31b).

The forward Euler integration of the advection terms imposes rather severe restrictions on the allowable timestep. For this reason, alternative explicit integration schemes have been used repeatedly (Wesseling (2001)). Many authors have used multilevel schemes, such as the second-order Adams–Bashforth scheme. The problem with schemes of this kind is that they use the values at the current and previous timestep, which makes them awkward in the context of adaptive refinement, moving meshes and local or global remeshing. Single step schemes are therefore preferable. Lax–Wendroff or Taylor–Galerkin schemes offer such a possibility, but in this case the result of steady-state calculations depends (albeit weakly) on the timestep (or equivalently the Courant number) chosen. This leads us to single step schemes whose steady-state result does not depend on the timestep. Schemes of this kind (explicit advection with a variety of schemes, implicit diffusion, pressure-Poisson equation for the pressure increments) have been widely used (Bell *et al.* (1989), Löhner (1990a), Martin and Löhner (1992), Bell and Marcus (1992), Alessandrini and Delhommeau (1996), Kallinderis and Chen (1996), Ramamurti and Löhner (1996), Löhner *et al.* (1999), Takamura *et al.* (2001), Karbon and Kumarasamy (2001), Codina (2001), Li *et al.* (2002), Karbon and Singh (2002)).

The resulting large, but symmetric system of equations given by (11.31a, b) and (11.32a, b) are of the form

$$\mathbf{K}\mathbf{u} = \mathbf{r}. \tag{11.33}$$

For large 3-D grids, iterative solvers are well suited for such systems. Preconditioned conjugate gradient (PCG) solvers (Saad (1996)) are most often used to solve (11.33). For isotropic grids, simple diagonal preconditioning is very effective. For highly stretched RANS grids, linelet preconditioning has proven superior (Martin and Löhner (1992), Soto *et al.* (2003)). In principle, multigrid solvers (Rhie (1986), Waltz and Löhner (2000), Trottenberg *et al.* (2001), Wesseling (2004)) should be the most efficient ones.

11.5. Temporal discretization: implicit schemes

Using the notation

$$u^{\Theta} = (1 - \Theta)u^n + \Theta u^{n+1}, \tag{11.34}$$

which implies

$$u^{n+1} - u^n = \frac{u^{\Theta} - u^n}{\Theta}, \tag{11.35}$$

an implicit timestepping scheme may be written as follows:

$$\frac{\mathbf{v}^{\theta} - \mathbf{v}^n}{\theta \Delta t} + \mathbf{v}^{\Theta}\nabla\mathbf{v}^{\Theta} + \nabla p^{\Theta} = \nabla\mu\nabla\mathbf{v}^{\Theta}, \tag{11.36}$$

$$\nabla \cdot \mathbf{v}^{\Theta} = 0. \tag{11.37}$$

Following similar approaches for compressible flow solvers (Alonso *et al.* (1995)), this system can be interpreted as the steady-state solution of the pseudo-time system:

$$\mathbf{v}^{\theta}_{,\tau} + \mathbf{v}^{\Theta}\nabla\mathbf{v}^{\Theta} + \nabla p^{\Theta} = \nabla\mu\nabla\mathbf{v}^{\Theta} - \frac{\mathbf{v}^{\theta} - \mathbf{v}^n}{\theta \Delta t}, \tag{11.38}$$

$$\nabla \cdot \mathbf{v}^{\Theta} = 0. \tag{11.39}$$

Observe that the only difference between (11.35) and (11.36) and the original incompressible Navier–Stokes equations given by (11.1) and (11.2) is the appearance of new source terms. These source terms are pointwise dependent on the variables being integrated ($\mathbf{v}$), and can therefore be folded into the LHS for explicit timestepping without any difficulty. The idea is then to march (11.35) and (11.36) to a steady state in the pseudo-time τ using either an explicit-advection projection scheme or an explicit artificial compressibility scheme using local timesteps.

11.6. Temporal discretization of higher order

The scheme given by (11.32a,b) is, at best, of second order in time. This may suffice for many applications, but for an accurate propagation of vortices in highly transient flows, temporal integration schemes of higher order will be advantageous. An interesting alternative is to integrate with different timestepping schemes the different regimes of flows with highly variable cell Reynolds number

$$Re_h = \frac{\rho|\mathbf{v}|h}{\mu}, \tag{11.40}$$

where h is the mesh size. For the case $Re_h < 1$ (viscous dominated), the accuracy in time of the advective terms is not so important. However, for $Re_h > 1$ (advection dominated), the advantages of higher-order time-marching schemes for the advective terms are considerable, particularly if one considers vortex transport over large distances. Dahlquist's (1963) theorem states that there exists no unconditionally stable linear multistep scheme that is of order higher than two (this being the Crank–Nicholson scheme). However, explicit schemes of the Runge–Kutta type can easily yield higher-order timestepping. A k-step, time-accurate Runge–Kutta scheme or order k for the advective parts may be written as

$$\mathbf{v}^i = \mathbf{v}^n + \alpha^i \gamma \Delta t(-\mathbf{v}^{i-1} \cdot \nabla \mathbf{v}^{i-1} - \nabla p^n + \nabla \mu \nabla \mathbf{v}^{i-1}), \quad i = 1, k-1; \tag{11.41}$$

$$\left[\frac{1}{\Delta t} - \theta \nabla \mu \nabla\right](\mathbf{v}^k - \mathbf{v}^n) + \mathbf{v}^{k-1} \cdot \nabla \mathbf{v}^{k-1} + \nabla p^n = \nabla \mu \nabla \mathbf{v}^{k-1}. \tag{11.42}$$

Here, the α^i are the standard Runge–Kutta coefficients $\alpha^i = 1/(k+1-i)$. As compared to the original scheme given by (11.32a), the $k-1$ stages of (11.41) may be seen as a predictor (or replacement) of $\mathbf{v}^n$ by $\mathbf{v}^{k-1}$. The original RHS has not been modified, so that at the steady state $\mathbf{v}^n = \mathbf{v}^{k-1}$, preserving the requirement that the steady state be independent of the timestep Δt. The factor γ denotes the local ratio of the stability limit for explicit timestepping for the viscous terms versus the timestep chosen. Given that the advective and viscous timestep limits are proportional to

$$\Delta t_a \approx \frac{h}{|\mathbf{v}|}, \quad \Delta t_v \approx \frac{\rho h^2}{\mu}, \tag{11.43}$$

one immediately obtains

$$\gamma = \frac{\Delta t_v}{\Delta t_a} \approx \frac{\rho|\mathbf{v}|h}{\mu} \approx Re_h, \tag{11.44}$$

or, in its final form,

$$\gamma = \min(1, Re_h). \tag{11.45}$$

In regions away from boundary layers, this factor is $O(1)$, implying that a high-order Runge–Kutta scheme is recovered. Conversely, for regions where $Re_h = O(0)$, the scheme reverts to the original one (equation (11.32a)). Note also that the very tempting option of ignoring the pressure and viscous terms in (11.41) leads to steady-state results that are not independent of the timestep.

Besides higher accuracy, an important benefit of explicit multistage advection schemes is the larger timestep one can employ. The increase in allowable timestep is roughly proportional to the number of stages used. This has been exploited extensively for compressible flow simulations (Jameson *et al.* (1981)). Given that for an incompressible solver of the projection type given by (11.32) most of the CPU time is spent solving the pressure-Poisson system (11.32b), the speedup achieved is also roughly proportional to the number of stages used. For further details, as well as timings and examples, see Löhner (2004).

11.7. Acceleration to the steady state

For steady flows, the use of a time-accurate scheme with uniform timestep Δt in the domain will invariably lead to slow convergence. In order to obtain steady results faster, a number of possibilities can be explored. Among those that have been reported in the literature, the following have proven the most successful:

- local timesteps;
- reduced iteration for the pressure;
- substepping for the advection terms;
- implicit treatment of the advection terms; and
- fully implicit treatment of advection, diffusion and pressure.

The main features of these are reviewed in what follows.

11.7.1. LOCAL TIMESTEPPING

Faster convergence to the steady state may be achieved by employing local timesteps. Given that the results obtained by the schemes used do not depend on the timestep, this can be readily done. One simply defines a separate timestep for each gridpoint and marches in time until a steady solution is reached. Local timestepping works best for problems that exhibit large variation of grid size and velocity.

11.7.2. REDUCED PRESSURE ITERATIONS

The most time-consuming part of projection schemes is the solution of the pressure-Poisson equation at every timestep. Recall that this is the equation that defines the pressure and establishes a divergence-free state at the next timestep. If one is only interested in the steady result, obtained after many timesteps, then the maintenance of an exact divergence-free state at every intermediate timestep is not required. Therefore, one can use a less stringent convergence criterion for the pressure-Poisson solver, saving a considerable amount of CPU time. Extensive experimentation with this option indicates mixed results. For inviscid flows (Euler equations, isotropic grids), it works well. However, for the more demanding high-Reynolds-number cases (separation, highly anisotropic grids), it has proven difficult to define reliable convergence criteria.

11.7.3. SUBSTEPPING FOR THE ADVECTION TERMS

This option was already treated above (see equations (11.41) and (11.42)). The speedup achieved is roughly proportional to the stages used.

11.7.4. IMPLICIT TREATMENT OF THE ADVECTION TERMS

Any explicit integration of the advective terms implies that information can only travel at most one element per timestep. In order to allow for a faster transfer of information and larger timesteps, the advective terms have to be integrated implicitly (Soto *et al.* (2001)). Equation (11.32a) then becomes

$$\left[\frac{1}{\Delta t} + \mathbf{v}^* \cdot \nabla - \nabla\mu\nabla\right](\mathbf{v}^* - \mathbf{v}^n) + \mathbf{v}^n \cdot \nabla\mathbf{v}^n + \nabla p^n = \nabla\mu\nabla\mathbf{v}^n, \tag{11.46}$$

leading to a non-symmetric system of equations of the form

$$\mathbf{A}\Delta\mathbf{v} = \mathbf{r}. \tag{11.47}$$

This may be rewritten as

$$\mathbf{A} \cdot \Delta\mathbf{v} = (\mathbf{L} + \mathbf{D} + \mathbf{U}) \cdot \Delta\mathbf{v} = \mathbf{r}, \tag{11.48}$$

where **L**, **D** and **U** denote the lower, diagonal and upper diagonal entries of **A**. Classic relaxation schemes to solve this system of equations include:

(a) Gauss–Seidel, given by

$$(\mathbf{L} + \mathbf{D}) \cdot \Delta\mathbf{v}^1 = \mathbf{r} - \mathbf{U} \cdot \Delta\mathbf{v}^0, \tag{11.49}$$

$$(\mathbf{D} + \mathbf{U}) \cdot \Delta\mathbf{v} = \mathbf{r} - \mathbf{L} \cdot \Delta\mathbf{v}^1; \tag{11.50}$$

(b) lower-upper symmetric Gauss–Seidel (LU-SGS), given by

$$(\mathbf{L} + \mathbf{D}) \cdot \mathbf{D}^{-1} \cdot (\mathbf{D} + \mathbf{U}) \cdot \Delta\mathbf{v} = \mathbf{r}. \tag{11.51}$$

These relaxation schemes have been optimized over the years, resulting in very efficient edge-based compressible flow solvers (Luo *et al.* (1998, 1999, 2000), Sharov *et al.* (2000a)). Key ideas include:

- using the spectral radius ρ_A of **A** for the diagonal entries **D**; for the advection case, $\rho_A = |\mathbf{v}|$, resulting in

$$\mathbf{D} = \left[\frac{1}{\Delta t}\mathbf{M}_l^i - 0.5 \sum \mathbf{C}^{ij}|\mathbf{v}|_{ij} + \sum \mathbf{k}^{ij}\right]\mathbf{I}, \tag{11.52}$$

  where **C**, **k** denote the edge coefficients for the advective and viscous fluxes and $\mathbf{M}_l^i$ the lumped mass matrix at node i;

- replacing

$$\mathbf{A}\cdot\Delta\mathbf{v}\approx\Delta\mathbf{F}, \tag{11.53}$$

resulting in

$$\Delta\mathbf{F}=\mathbf{F}(\mathbf{v}+\Delta\mathbf{v})-\mathbf{F}(\mathbf{v}). \tag{11.54}$$

The combined effect of these simplifications is a family of schemes that are matrix-free, require no extra storage as compared to explicit schemes and (due to lack of limiting) per relaxation sweep are faster than conventional explicit schemes. For the LU-SGS scheme, each pass over the mesh proceeds as follows:

- forward sweep,

$$\Delta\hat{\mathbf{v}}^i=\mathbf{D}^{-1}\left[\mathbf{r}^i-0.5\sum_{j<i}\mathbf{C}^{ij}\cdot(\Delta\hat{\mathbf{F}}_{ij}-|\mathbf{v}|_{ij}\Delta\hat{\mathbf{v}}_j)+\sum_{j<i}\mathbf{k}^{ij}\Delta\hat{\mathbf{v}}_j\right]; \tag{11.55}$$

- backward sweep,

$$\mathbf{r}=\mathbf{D}\cdot\Delta\hat{\mathbf{v}}, \tag{11.56}$$

$$\Delta\mathbf{v}^i=\mathbf{D}^{-1}\left[\mathbf{r}^i-0.5\sum_{j>i}\mathbf{C}^{ij}\cdot(\Delta\mathbf{F}_{ij}-|\mathbf{v}|_{ij}\Delta\mathbf{v}_j)+\sum_{j>i}\mathbf{k}^{ij}\Delta\hat{\mathbf{v}}_j\right]. \tag{11.57}$$

Luo *et al.* (1998) have shown that no discernable difference could be observed when taking central or upwind discretizations for $\Delta\mathbf{F}$. As the CPU requirements of upwind discretizations are much higher, all relaxation passes are carried out using central schemes. Given that the same loop structure (**L**, **D**, **U**) is required for both the Gauss–Seidel, the LU-SGS and the GMRES matrix–vector products, it is possible to write a single ‘sweep’ subroutine that encompasses all of these cases. The initialization of the Gauss–Seidel loop is accomplished with an LU-SGS pass.

11.8. Projective prediction of pressure increments

The solution of the Poisson equation given by (11.32b) is typically carried out with a PCG (Saad (1996)) solver, and consumes a considerable percentage of the overall computational effort of projection-type schemes. Any gain (e.g. in the form of a reduction in the number of iterations) will have an immediate impact on the overall execution speed of the solver. Equation (11.32b) results in a discrete system of the form

$$\mathbf{K}\cdot\Delta\mathbf{p}=\mathbf{r}. \tag{11.58}$$

One way to reduce the number of iterations required is to start with a value of $\Delta\mathbf{p}$ that is close to the solution. For time-accurate problems with constant timesteps an extrapolation from previous increments seems a reasonable proposition. However, experience (Fischer (1998)) indicates that this does not yield a reliable way of reducing the number of iterations. Most solvers tend to initialize the iterative solver for (11.58) with $\Delta\mathbf{p}=0$. The rationale given for this choice is that at a steady state $\Delta\mathbf{p}=0$, i.e. as the solution is converging this represents a good choice. On the other hand, it can be argued that the pressure increment between timesteps is similar (Fischer (1998)). If one considers the case of a vortex street behind a cylinder or a car, this is certainly the case, as many timesteps are required per shedding cycle. For this reason, it seems reasonable to seek an estimate of the starting value $\Delta\mathbf{p}$ based on the values obtained at previous timesteps. In what follows, the *basic assumption* is that $\mathbf{K}$ does not change in time. For many incompressible flow solvers this is indeed the case. Solvers that use some form of stabilization or consistent numerical fluxes (e.g. in the form of a fourth-order damping) do not fall under this category. For these cases it may be argued that $\mathbf{K}$ changes very little.

Denoting by $\Delta\mathbf{p}^i$, $\mathbf{r}^i$, $i = 1, l$ the values of the pressure increments and RHSs at previous timesteps $n - i$, we know that

$$\mathbf{K} \cdot \Delta\mathbf{p}^i = \mathbf{r}^i. \tag{11.59}$$

Given the new RHS $\mathbf{r}$, one can perform a least-squares approximation to it in the basis $\mathbf{r}^i$, $i = 1, l$,

$$(\mathbf{r} - \alpha_i \mathbf{r}^i)^2 \to \min, \tag{11.60}$$

which results in

$$\mathbf{A}\boldsymbol{\alpha} = \mathbf{s}, \quad A^{ij} = \mathbf{r}^i \cdot \mathbf{r}^j, \quad s^i = \mathbf{r}^i \cdot \mathbf{r}. \tag{11.61}$$

Having solved for the approximation coefficients α_i, one then estimates the start value $\Delta\mathbf{p}$ from

$$\Delta\mathbf{p} = \alpha_i \Delta\mathbf{p}^i. \tag{11.62}$$

In principle, the use of the RHSs $\mathbf{r}^i$, $i = 1, l$ as a basis may be numerically dangerous. After all, if any of these vectors are parallel, the matrix $\mathbf{A}$ is singular. One could instead perform a Gram–Schmidt orthogonalization. This option was invoked by Fischer (1998), who looked at a number of possible schemes to accelerate the convergence of iterative solvers using successive RHSs within the context of incompressible flow solvers based on spectral elements. However, this has not been found to be a problem in practice. The advantage of using simply the original RHSs is that the update of the basis is straightforward. An index is kept for the last entry in the basis, and the new entries at the end of the timestep are inserted at the position of the oldest basis vector. The storage requirements for this projective predictor scheme are rather modest: `2*npoin*nvecl`. One typically uses no more than four basis vectors, i.e. the storage is at most `8*npoin`. For further details, see Löhner (2005).

11.9. Examples

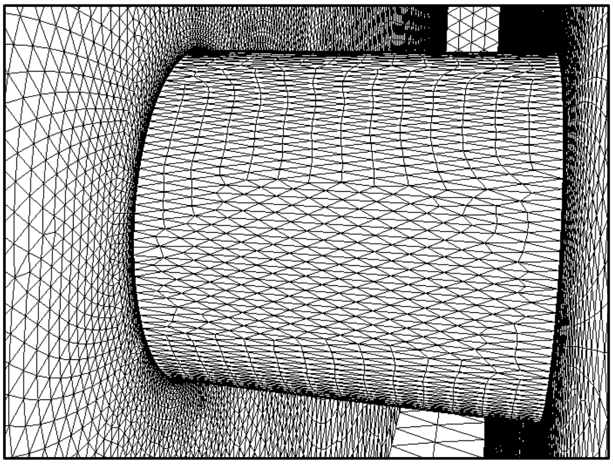
11.9.1. VON KARMAN VORTEX STREET

This is a well-known benchmark case (Schlichting (1979)). A circular cylinder is suspended in a uniform stream of incompressible fluid. The separation at the back of the cylinder generates the so-called von Karman vortex street, whose characteristics depend on the Reynolds number

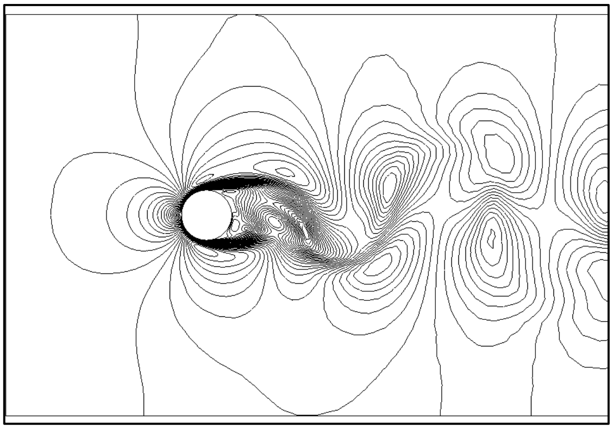
$$Re = \frac{\rho V_\infty D}{\mu},$$

where D denotes the diameter of the cylinder. A mesh of approximately 60 000 points and 300 000 elements, with special placement of points in the vicinity of the cylinder, was used for the simulation. The parameters were chosen such that the resulting Reynolds number was

$Re = 190$. The results were obtained by the incremental projection scheme given by (11.32), with both explicit and implicit time integration of the advective terms. The advection operator was treated with edge-based upwinding, and the divergence operator with the consistent numerical flux given by (11.24). Figures 11.2(a)–(b) show the surface grid and the absolute value of the velocity in a cut plane. The lift of the cylinder as a function of time for the original explicit-advection projection scheme, as well as the implicit-advection with implicitness parameters $\Theta = 0.50$, 0.60, 1.00 is displayed in Figure 11.2(c). Observe that the Strouhal number obtained is approximately $S = 0.2$ for all cases, in good agreement with experiments (Schlichting (1979)). However, the lift changes markedly. Choosing $\Theta = 1.0$ results in a lift force that is only 50% that of the correct solution. For $\Theta = 0.50$ some oscillations appear. This is to be expected, as this is the limit of unconditionally stable schemes.

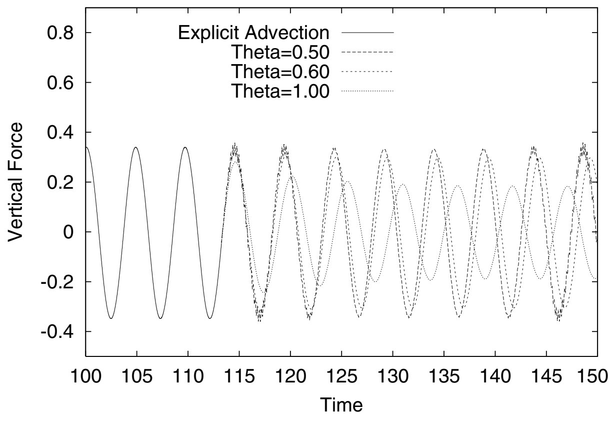


(a)

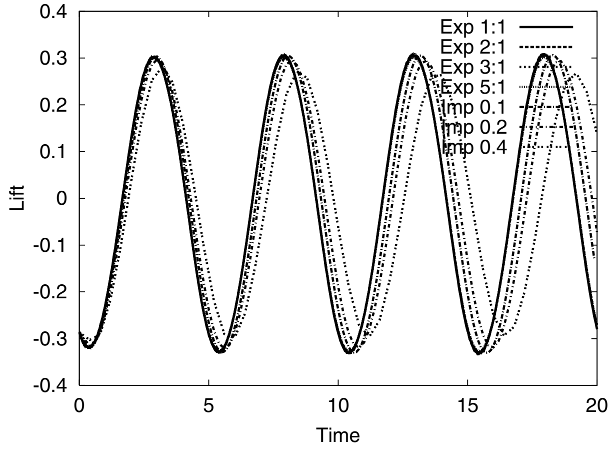


(b)

Figure 11.2. von Karman vortex street: (a) surface mesh; (b) Abs(velocity) in plane; (c), (d) vertical forces



(c)



(d)

Figure 11.2. Continued

The run was repeated with a number of timestepping schemes. The lift of the cylinder as a function of time for the original explicit-advection projection scheme, the 2/3/5-step explicit-advection projection scheme, as well as the implicit-advection Cranck–Nicholson ($\theta = 0.50$) scheme with different timesteps is displayed in Figure 11.2(d). For the implicit timestepping scheme, the pseudo-steady-state system given by (11.38) and (11.39) was solved using a local Courant number of $C = 4.0$, implicit advection and SGS relaxation. Typically, 10–20 SGS iterations per timestep were required for convergence. Observe that for the schemes that employ a reasonable timestep, and in particular all the explicit-advection multistep schemes, the results are almost identical. Not surprisingly, as the timestep is increased, the implicit schemes remain stable, but the solution deteriorates. Recall that for $\Delta t = 0.4$, a particle in the free stream crosses the cylinder in 2.5 timesteps(!). The timings recorded are summarized in Table 11.2. Note that the timestep for the explicit-advection schemes is approximate.

Table 11.2. von Karman vortex street

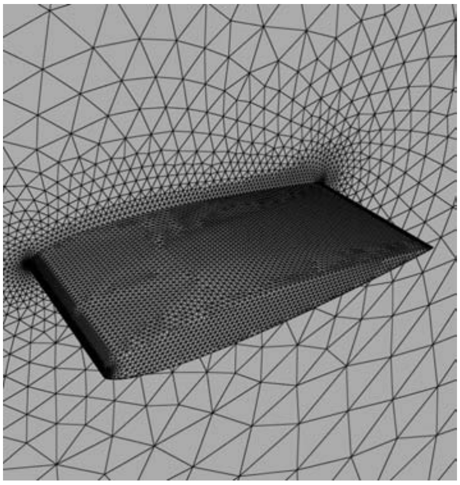
Scheme	Δt	`ntime`	CPU (s)	Speedup
Ex 1 0.1	0.002	9961	12 929	1.00
Ex 2 0.4	0.008	2490	4194	3.08
Ex 3 0.6	0.012	1660	3296	3.92
Ex 5 0.8	0.016	1245	3201	4.03
Ex 5 1.2	0.025	830	1546	8.36
Ex 5 1.6	0.033	623	1114	11.60
Ex 5 1.8	0.037	554	995	12.99
ImSGS (5)	0.1	200	3189	4.05
ImSGS (5)	0.2	100	1612	8.02
ImSGS (5)	0.4	50	962	13.43

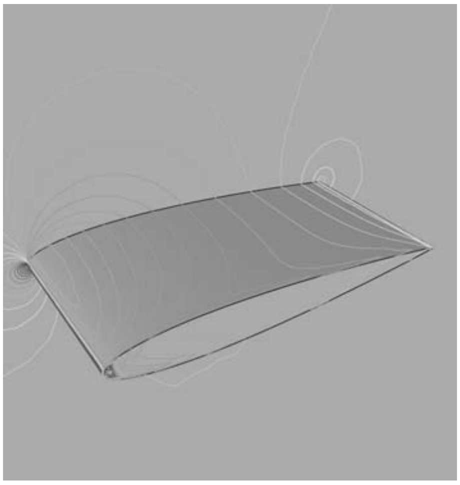
11.9.2. NACA0012 WING

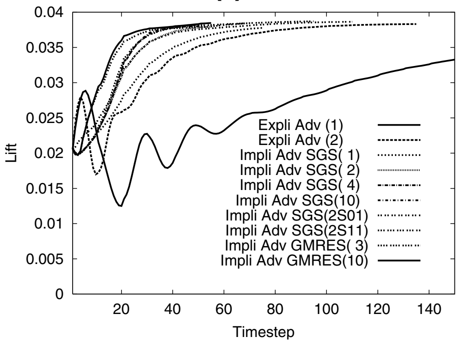
This classic example considers a NACA0012 wing at $\alpha = 5^\circ$ angle of attack. The aim of this test is to gauge the performance of the different schemes presented, as well as the projective pressure increment predictor for a steady, inviscid (Euler) case. Figures 11.3(a) and (b) show the surface mesh employed, as well as the surface pressures obtained. The mesh consisted of 68 000 points and 368 000 elements. With C denoting the Courant number, this problem was solved using:

- the standard explicit-advection projection scheme ($C = 0.1$) with full pressure solution every 20 timesteps, and partial pressure solution (max 20 PCG iterations) in between (Ex 1);
- the explicit-advection projection scheme with 2–5 substeps for advective prediction (Ex 2–5);
- the implicit-advection projection scheme ($C = 4.0$) with several SGS relaxations (ImSGS 1/2/4/10);
- a predictor-corrector scheme (not outlined above) based on the implicit-advection SGS solver (ImSGS 2 01/11); and
- the implicit-advection projection scheme ($C = 4.0$) with LU-SGS–GMRES solver for the advection, again with a different number of iterations per timestep (ImGMRES 3/10).

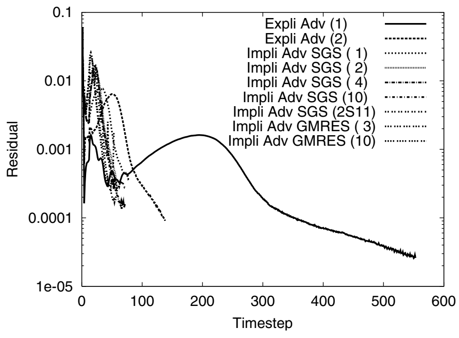
Each one of these runs was considered to be converged when the change in lift, normalized by the Courant number, was below $t_l = 10^{-3}$ for five subsequent timesteps. It is often found that such a measure is a better indicator of convergence than residuals, as it allows the user to state clearly what accuracy is desired. Figures 11.3(c) and (d) show the convergence history for the lift and residuals, respectively. One can see that the implicit-advection schemes converge significantly faster. The timings obtained are summarized in Table 11.3, and indicate that this faster convergence also translates into a significant reduction in CPU requirements. Note that the multistep explicit-advection schemes also yield a surprisingly high speedup,


(a)

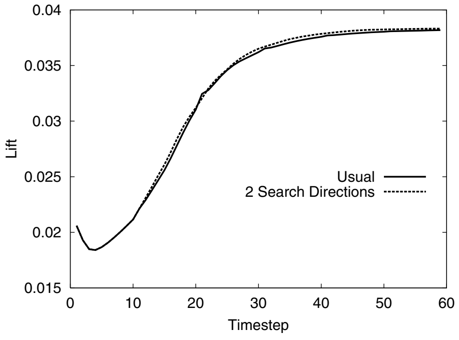

(b)



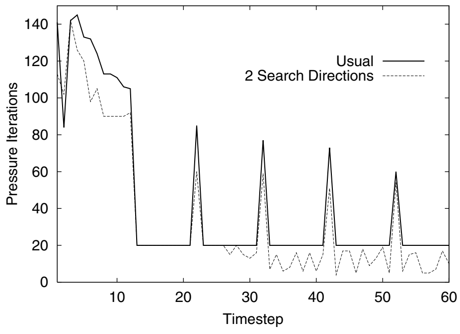
(c)



(d)



(e)



(f)

Figure 11.3. NACA 0012: (a) surface mesh; (b) pressure; convergence history for (c) lift and (d) residuals; (e) lift convergence; (f) pressure iterations

but fall short of the speedups obtained for the low-relaxation SGS schemes. The LU-SGS–GMRES schemes with a low number of search directions are also competitive. However, increasing the number of search directions has a counterproductive effect.

Table 11.3. NACA-0012

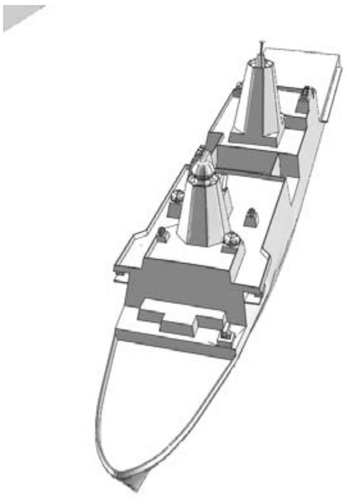
Scheme	`ntime`	CPU (s)	Speedup
Ex 1 (0.1)	540	579	1.00
Ex 2 (0.4)	135	193	3.00
Ex 3 (0.6)	90	157	3.69
Ex 5 (0.8)	70	166	3.48
ImSGS (1)	75	172	3.36
ImSGS (2)	55	142	4.07
ImSGS (4)	70	188	3.08
ImSGS (10)	51	532	1.09
ImSGS 2 (01)	110	556	1.04
ImSGS 2 (11)	50	280	2.07
ImGMRES (3)	70	216	2.68
ImGMRES (10)	55	772	0.75

The same case was then re-run using the projective prediction of pressure increments with the implicit LU-SGS/GS scheme for the advective prediction. Local timesteps were employed with a Courant number based on the advective terms of $C = 5$. Given that the mesh is isotropic, a diagonal preconditioner for the pressure-Poisson equation was used. As the flow is started impulsively, a divergence cleanup (first three iterations) precedes the advancement of the flowfield. After the residuals had converged by an order of magnitude, the number of pressure iterations was reduced artificially to $n_p = 20$, and the relative tolerance for residual convergence was increased to $tol_p = 10^{-2}$. Every tenth timestep, the residual convergence was lowered to the usual value of $tol_p = 10^{-4}$ in order to obtain a divergence-free flowfield. This procedure has been found to work well for inviscid, steady flows, reducing CPU requirements considerably as compared to keeping the relative tolerance constant throughout the run. The case was run for 60 steps, which was sufficient to lower the relative change in lift forces to below $tol_f = 10^{-3}$. The convergence of the lift may be seen in Figure 11.3(e). The required number of iterations per timestep is shown in Figure 11.3(f). Note that even for this steady, inviscid case the projective pressure predictor yields a considerable reduction in the number of pressure iterations.

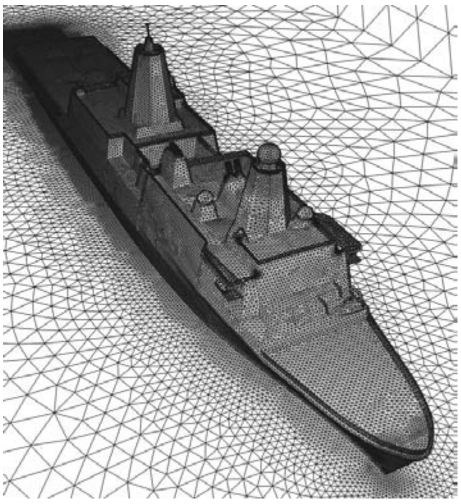
11.9.3. LPD-17 TOPSIDE FLOW STUDY

This case, taken from Camelli (2003), considers the external flow (so-called airwake studies) past a typical ship, and is included here as an example of CFD runs under time constraints. The geometry is illustrated in Figure 11.4(a). The dimensions of the ship are approximately as follows: length, 200 m; width, 30 m; height above the waterline, 50 m.

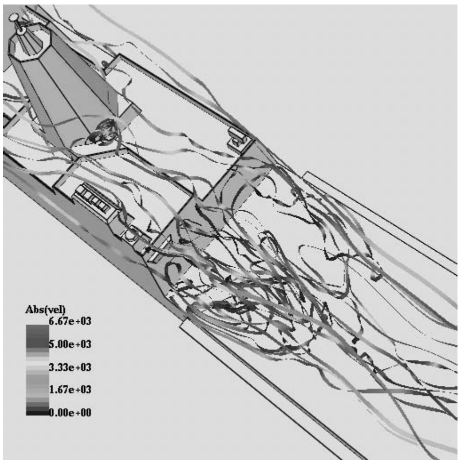
Two different grid resolutions were used. The coarser mesh had approximately 490 000 points and 2.7 million tetrahedra. The finer mesh had 990 000 points and 5.6 million tetrahedra. The surface mesh for the coarser mesh is shown in Figure 11.4(b). The Reynolds number based on the height of the ship is 5×10^7. Due to the presence of defined separation lines at ridges and corners and the resulting massive, unsteady separation, a VLES run using the Smagorinsky turbulence model (Smagorinsky (1963)) and an isotropic mesh seemed prudent. In the first part of the run, a pseudo-steady flow was established. After this state



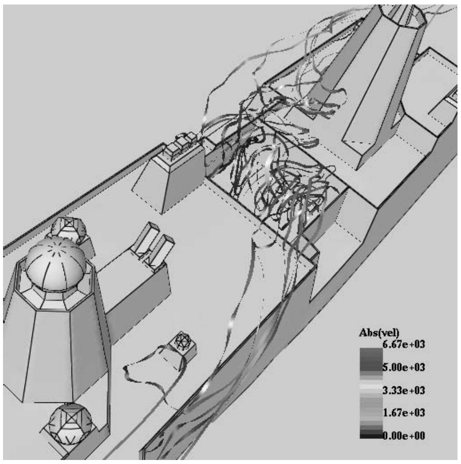
(a)



(b)



(c)

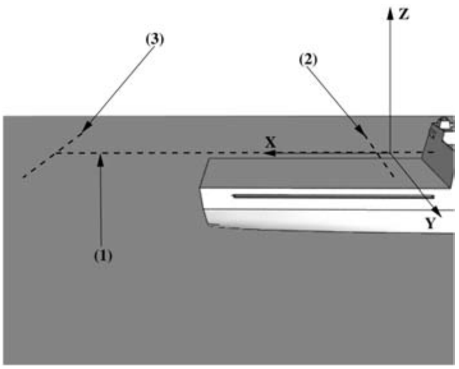


(d)

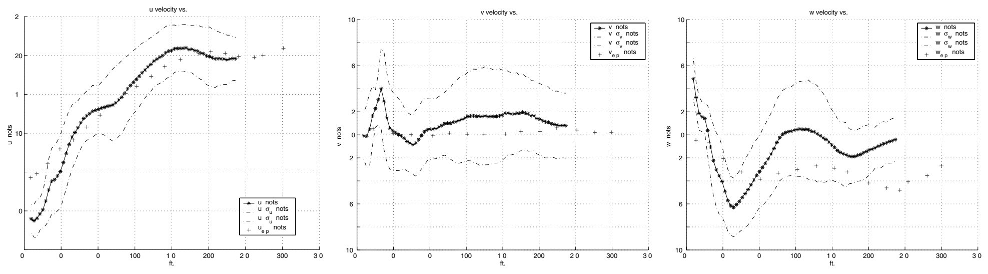
Figure 11.4. LPD-17: (a), (b) geometry; (c), (d) streamribbons; (e) location of experimental measurements (lines 1–3); (f) comparison of velocities for line 1; (g) line 2 and (h) line 3

was reached, the flow was integrated for 90 s of real time. The whole run, including the initialization part, took approximately two weeks on a non-dedicated 16-processor SGI 3800 shared-memory machine.

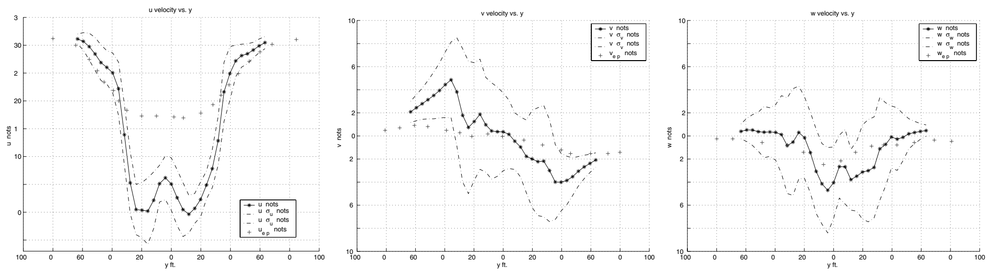
To indicate the complexity of the flows simulated, two groups of instantaneous streamribbons (shaded according to speed) are shown in Figures 11.4(c) and (d). The ribbons show a large recirculation above the landing deck, as well as a pronounced crossflow from port to starboard near the middle of the ship. Experimental velocity data from wind tunnel LDV measurements was available at the positions shown in Figure 11.4(e). The experimental velocities were averaged and compared with averaged numerical results (90 s period).



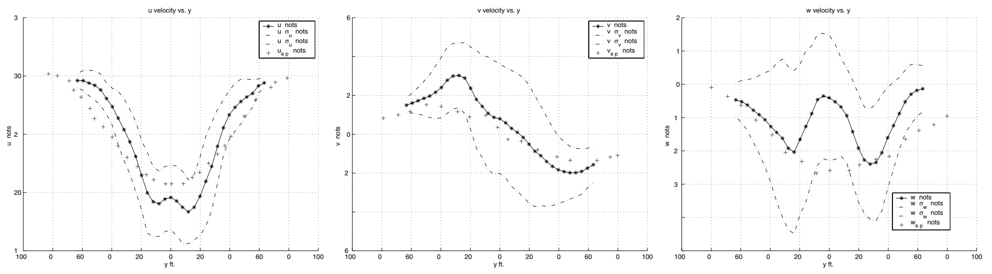
(e)



(f)

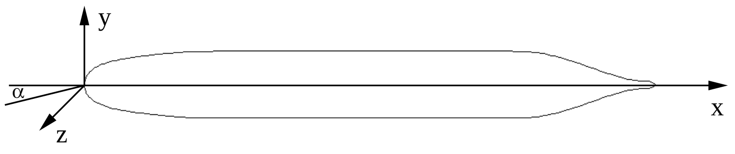


(g)

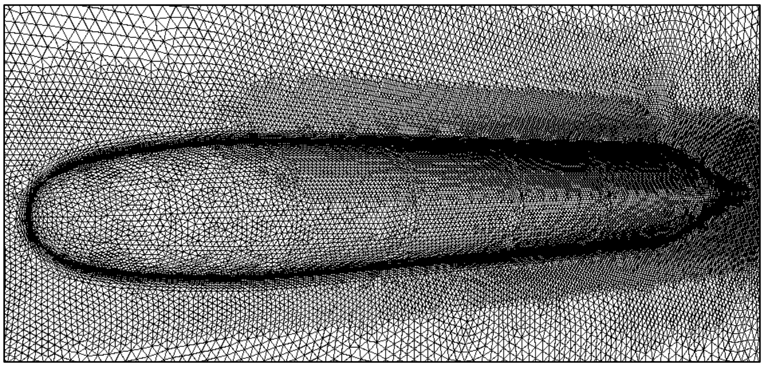


(h)

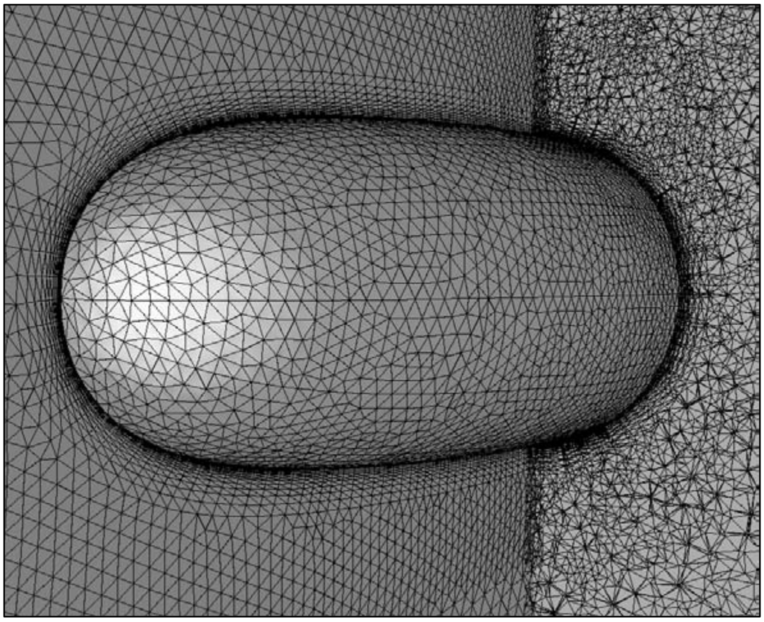
Figure 11.4. Continued



(a)

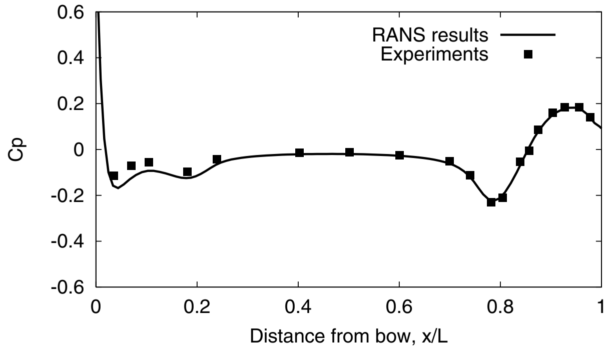


(b)

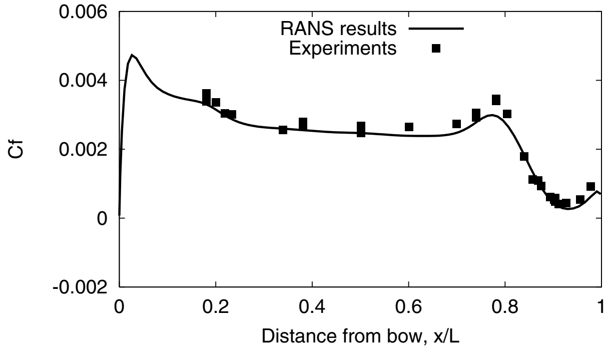


(c)

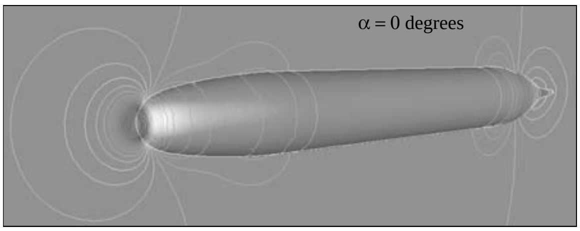
Figure 11.5. (a) Definition sketch of SUBOFF model without a fairwater; (b) surface mesh; (c) surface mesh and mesh in cut plane; comparison of computed and measured (d) pressure and (e) skin friction coefficients; (f) pressure and (g) shear vectors on the surface



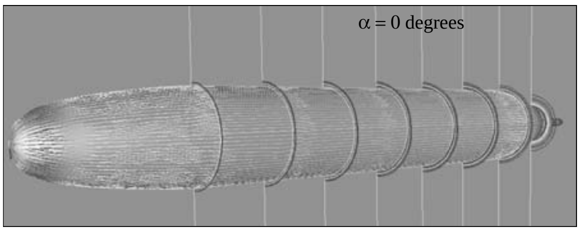
(d)



(e)



(f)



(g)

Figure 11.5. Continued

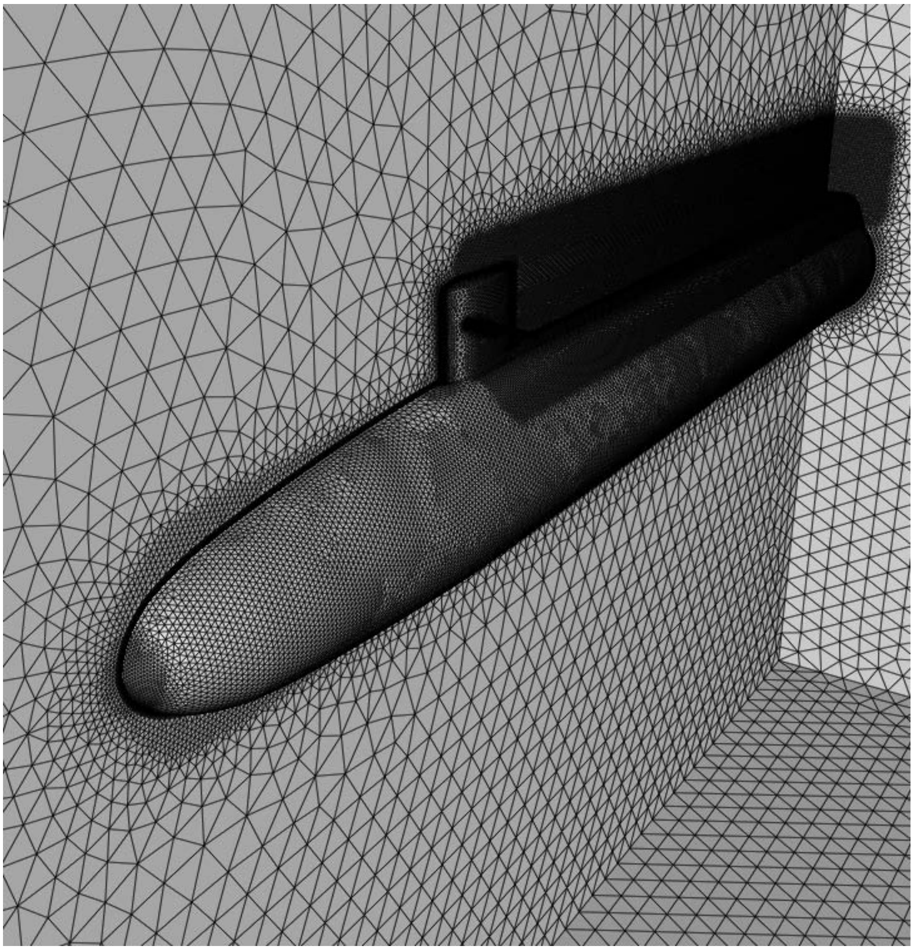


Figure 11.6. Submarine forebody: (a) surface mesh; (b) surface mesh (detail); (c) surface pressure, velocity

Figure 11.4(f)–(h) shows the comparison of computed and measured average velocities. The crosses indicate the experimental data, the stars the average of the numerical results and the dash-dotted lines the standard deviation for the numerical results. Given the coarseness of the mesh and the simple turbulence model, the numerical results agree surprisingly well with the experimental data. For further details, see Camelli *et al.* (2003).

11.9.4. DARPA SUBOFF MODEL

The flow around an appended underwater body is characterized by thick boundary layers, vortical flow structures generated by hull-appendage junctures and appendage turbulent wakes (Groves *et al.* (1998), Huang *et al.* (1992)). To understand these complex flow characteristics, a typical submarine configuration was designed for the DARPA SUBOFF

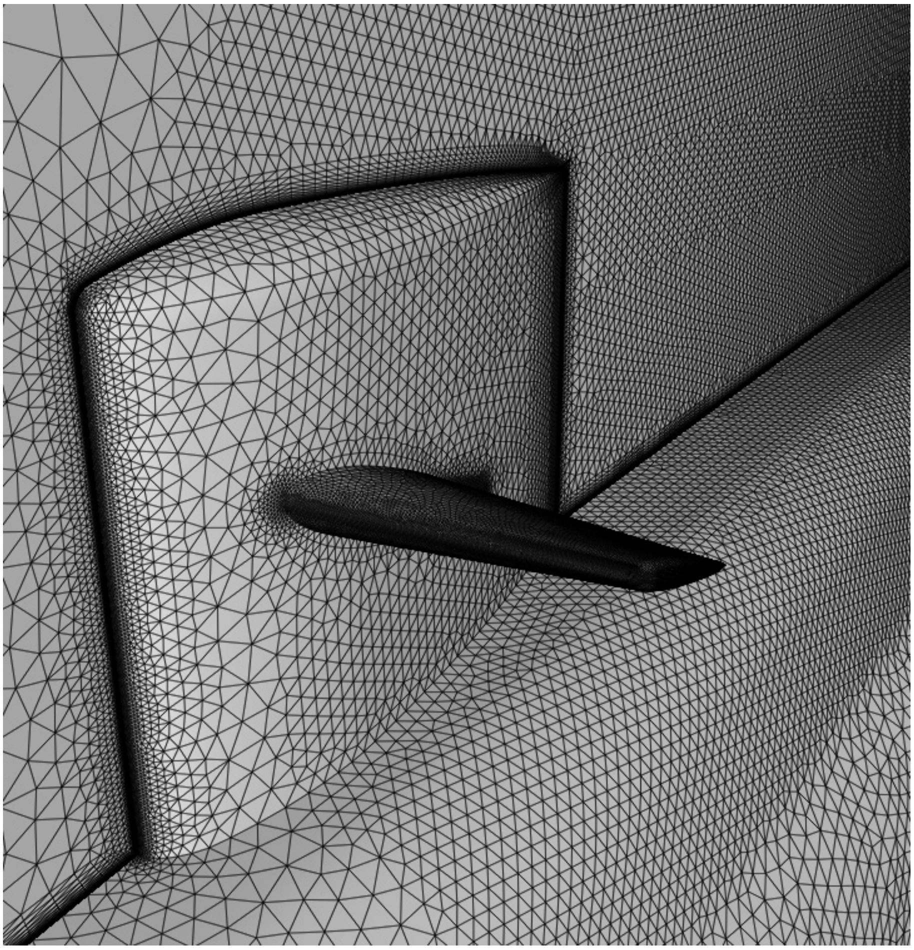
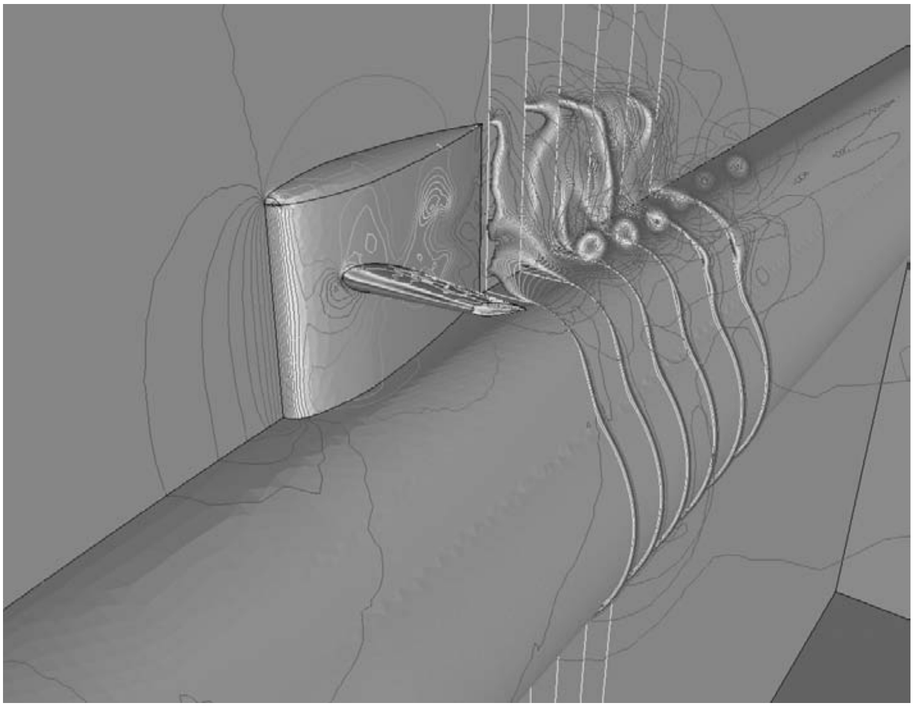

(b)

Figure 11.6. Continued

project (Groves *et al.* (1998)). The case considered here, taken from a comprehensive study (Yang and Löhner (2003)), is the SUBOFF model without a fairwater at zero degree incidence as sketched in Figure 11.5(a). The Reynolds number based on the body length is 1.2×10^7, and the effects of turbulence are modelled by the Baldwin–Lomax model.

The mesh of the computational domain consisted of 3.93 million elements and 683 000 points. The surface grid, shown in Figures 11.5(b) and (c), is typical of RANS calculations. In the proximity of the wall, the elements are highly anisotropic with extremely fine spacing normal to the wall. Away from the wall the mesh coarsens rapidly and becomes isotropic.

Figures 11.5(d) and (e) present a comparison of the computed pressure and skin friction coefficients along the upper meridian line of the hull with experimental data. One can see that the numerical predictions are in very good agreement with the experimental data for both pressure and skin friction coefficients. Figures 11.5(f) and (g) show the pressure and shear



(c)

Figure 11.6. Continued

vectors on the surface, as well as the contours of absolute velocity in a series of cut planes normal to the x-axis.

11.9.5. GENERIC SUBMARINE FOREBODY VORTEX FLOW STUDY

This case considers the flow past a submarine with a 42-feet diameter. The speed is 7 knots and the Reynolds number per foot is $Re = 1.1 \times 10^6$. Turbulence is simulated with the Baldwin–Lomax model (Baldwin and Lomax (1978)). The front half of the submarine is modelled in order to study the vortex shed by the sail.

The RANS mesh had approximately `nelem=6Mtets` elements, and was suitably refined in boundary-layer regions and the region behind the sail. Figures 11.6(a)–(c) show different views of the surface grid, as well as the surface pressure, together with contours of the absolute value of the velocities in six planes behind the sail. The development of a main tip vortex and several secondary vortices as well as boundary-layer detachment is clearly visible.

12 MESH MOVEMENT

In many instances, the fluid interacts with rigid or flexible bodies that are either totally submerged in it or partly wetted. These bodies will tend to react to the forces exerted by the fluid, producing a change of the fluid domain. Examples of this type of interaction are structures submerged in water, aeroelastic applications like flutter or buzz, or simply the movement of flags due to wind forces. Another possibility is the forced movement of a body through a flowfield. Examples that fall under this category are store separation for military aircraft, torpedo launch and trains entering tunnels. The important new phenomenon that has to be addressed carefully for these classes of problems is the change of the domain itself as the simulation proceeds. The present chapter discusses the changes required for the PDEs describing the fluid, the algorithms employed and the possibilities for mesh movement.

12.1. The ALE frame of reference

The most convenient form to treat problems with changing domains is to re-write the equations describing the fluid in an arbitrary Lagrangian Eulerian (ALE) frame of reference. Given a mesh velocity field $\mathbf{w} = (w^x, w^y, w^z)$, the Euler equations become

$$\begin{Bmatrix} \rho \\ \rho u^x \\ \rho u^y \\ \rho u^z \\ \rho e \end{Bmatrix}_{,t} + \begin{Bmatrix} (u^i - w^i)\rho \\ (u^i - w^i)\rho u^x + p \\ (u^i - w^i)\rho u^y \\ (u^i - w^i)\rho u^z \\ (u^i - w^i)\rho e + u^i p \end{Bmatrix}_{,i} = -\nabla \cdot \mathbf{w} \begin{Bmatrix} \rho \\ \rho u^x \\ \rho u^y \\ \rho u^z \\ \rho e \end{Bmatrix}. \tag{12.1}$$

Observe that in the case of no element movement ($\mathbf{w} = \mathbf{0}$) we recover the usual Eulerian conservation law form of the Euler equations. If, however, the elements move with the particle velocity ($\mathbf{w} = \mathbf{v}$), we recover the Lagrangian form of the equations of motion. From the numerical point of view, (12.1) implies that all that is required when going from an Eulerian frame to an ALE frame is a modified evaluation of the fluxes on the LHS and the additional evaluation of source terms on the RHS. After spatial and temporal discretization, the resulting system of equations for explicit timestepping is of the form

$$\mathbf{M}_c|^n \Delta\mathbf{u} = \mathbf{r} + \mathbf{s}, \tag{12.2}$$

where $\mathbf{r}$ and $\mathbf{s}$ denote the contributions of the advective fluxes and source terms, respectively. These source terms, and in particular the divergence of $\mathbf{w}$, reflect the variation of element size in time. One can avoid the explicit evaluation of these source terms by rewriting the Euler equations in an integral form as

$$\frac{d}{dt}\int_{\Omega(t)} \mathbf{u}\, d\Omega + \int_{\Omega(t)} \nabla \cdot \mathbf{F}\, d\Omega = 0. \tag{12.3}$$

Applied Computational Fluid Dynamics Techniques: An Introduction Based on Finite Element Methods, Second Edition.
Rainald Löhner © 2008 John Wiley & Sons, Ltd. ISBN: 978-0-470-51907-3

After discretization, the system of equations now assumes the form

$$\mathbf{M}_c|^{n+1}\mathbf{u}^{n+1} - \mathbf{M}_c|^{n}\mathbf{u}^{n} = \mathbf{r}. \tag{12.4}$$

This may be rewritten as

$$\mathbf{M}_c|^{n+1}\Delta\mathbf{u} = \mathbf{r} - (\mathbf{M}_c|^{n+1} - \mathbf{M}_c|^{n}) \cdot \mathbf{u}^{n}, \tag{12.5}$$

where the correlation between the terms in (12.2) and (12.5) now becomes apparent.

12.1.1. BOUNDARY CONDITIONS

When imposing the boundary conditions for the velocities at solid walls, we need to take the velocity of the surface **w** into consideration. For Navier–Stokes solvers, this results in the condition

$$\Delta\rho\mathbf{v} = \Delta\rho\mathbf{w}, \tag{12.6}$$

at the surface. For Euler solvers, only the velocity normal to the moving surface should vanish. Denoting the predicted momentum at the surface as $\Delta\rho\mathbf{v}^*$, we can decompose it as follows:

$$\Delta\rho\mathbf{v}^* = \Delta[\rho(\mathbf{w} + \alpha\mathbf{t} + \beta\mathbf{n})], \tag{12.7}$$

where **t** and **n** are the tangential and normal vectors, respectively. The desired momentum at the new timestep should, however, have no normal velocity component ($\beta = 0$) and it has the form

$$\Delta\rho\mathbf{v}^{n+1} = \Delta[\rho(\mathbf{w} + \alpha\mathbf{t})]. \tag{12.8}$$

Combining (12.7) and (12.8), we obtain the two following cases.

(a) Given **t**:

$$\Delta\rho\mathbf{v}^{n+1} = \Delta\rho\mathbf{w} + [(\Delta\rho\mathbf{v}^* - \Delta\rho\mathbf{w}) \cdot \mathbf{t}] \cdot \mathbf{t}. \tag{12.9}$$

(b) Given **n**:

$$\Delta\rho\mathbf{v}^{n+1} = \Delta\rho\mathbf{v}^* - [(\Delta\rho\mathbf{v}^* - \Delta\rho\mathbf{w}) \cdot \mathbf{n}] \cdot \mathbf{n}. \tag{12.10}$$

12.2. Geometric conservation law

The motion of the mesh should, ideally, not influence the solution. In particular, if a steady flowfield is given, any arbitrary mesh motion should not create changes in the solution. A straightforward implementation of any of the forms of the ALE equations described above will not guarantee this. The only way to satisfy rigorously this requirement is by solving numerically what has been called the geometric conservation law (Batina (1990a), Lesoinne and Farhat (1996)) in order to obtain the new mass matrix at timestep $n + 1$. If we assume a constant density $\rho = 1$, then the continuity equation in the ALE frame of reference becomes

$$\mathbf{M}_c|^{n+1} - \mathbf{M}_c|^{n} + \Delta t \int_{\Omega^n} N^i \nabla \cdot (\mathbf{v} - \mathbf{w})\, d\Omega = 0. \tag{12.11}$$

For consistency, the time-advancement scheme used in (12.11) should be the same as the one used for the full system of Euler/Navier–Stokes equations. The differences between solving for the new mass matrices using the geometric conservation law or by straightforward precalculation of the geometric parameters at the new timestep are very often vanishingly small (Batina (1990a)). For this reason, many codes omit the use of geometric conservation law updates of the mass-matrix. However, it is advisable to use a geometric conservation law update whenever the mesh motion is severe or the mesh motion approaches the velocity of the fluid. This may be the case for some store separation problems, particularly in the later stages of the launch (Farhat *et al.* (1998), Sorensen *et al.* (2003)).

12.3. Mesh movement algorithms

An important question from the point of view of mesh distortion and remeshing requirements is the algorithm employed to move the mesh. Given that the mesh velocity on the moving surfaces of the computational domain is prescribed,

$$\mathbf{w}|_{\Gamma_b} = \mathbf{w}_b, \tag{12.12}$$

and, at a certain distance from these moving surfaces, as well as on all the remaining surfaces, the mesh velocity vanishes,

$$\mathbf{w}|_{\Gamma_0} = 0, \tag{12.13}$$

the question is how to obtain a mesh velocity field **w** in such a way that element distortion is minimized (see Figure 12.1). A number of algorithms have been proposed. They may be grouped together into the following families:

(a) smoothing the velocity field;

(b) smoothing the coordinates; and

(c) prescribing analytically the mesh velocity.

The possibilities for each of these families are described in what follows.

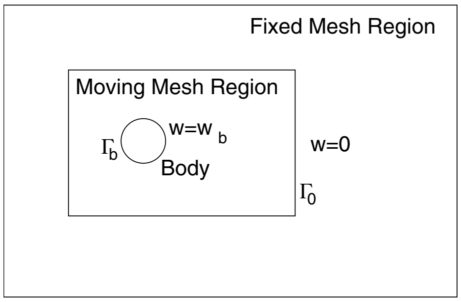


Figure 12.1. Mesh movement

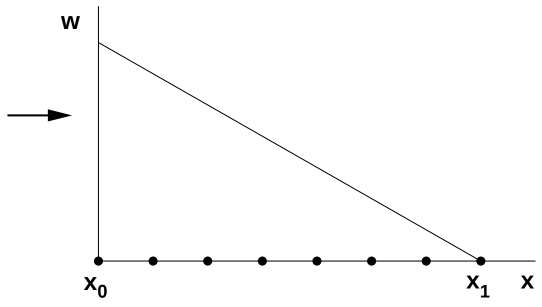


Figure 12.2. 1-D mesh movement

12.3.1. SMOOTHING OF THE VELOCITY FIELD

In this case, the mesh velocity is smoothed, based on the exterior boundary conditions given by (12.12) and (12.13). The aim, as stated before, is to obtain a mesh velocity field **w** so that element distortion is minimized. Consider for the moment the 1-D situation displayed in Figure 12.2.

At the left end of the domain, the mesh velocity is prescribed to be w_0. At the right end, the mesh velocity vanishes. If the mesh velocity decreases *linearly*, i.e.

$$\frac{\partial w}{\partial x} = g_w, \tag{12.14}$$

then the elements will maintain their initial size ratios. This is because, for any two elements, the change in size Δh during one timestep is given by

$$\Delta h = (w_2 - w_1)\Delta t = \Delta w \Delta t. \tag{12.15}$$

This implies that for the size ratio of any two elements i, j we obtain

$$\left.\frac{h_i}{h_j}\right|^{n+1} = \frac{h_i + \Delta w_i \Delta t}{h_j + \Delta w_j \Delta t} = \frac{h_i + g_w h_i \Delta t}{h_j + g_w h_j \Delta t} = \left.\frac{h_i}{h_j}\right|^{n}. \tag{12.16}$$

This implies that all elements in the regions where mesh velocity is present will be deformed in roughly the same way (think of 3-D situations with rotating or tumbling bodies immersed in the flowfield). Solutions with constant gradients are reminiscent of the solutions to the Laplace operator. Indeed, for the general case, the mesh velocity is obtained by solving

$$\nabla \cdot (\mathbf{k} \cdot \nabla \mathbf{w}) = 0, \tag{12.17}$$

with the Dirichlet boundary conditions given by (12.12) and (12.13). This system is again discretized using finite elements (Baum *et al.* (1994), Löhner and Yang (1996)) or finite volumes. The edge-based data structures treated in Chapter 10 can be used advantageously in this context. The resulting system of equations may be solved via relaxation as

$$C^{ii} \Delta \mathbf{w}^i = -\Delta t K^{ij} (\mathbf{w}^i - \mathbf{w}^j), \tag{12.18}$$

where

$$C^{ii} = \sum_{i \neq j} |K^{ij}|, \tag{12.19}$$

or, alternatively, by a conjugate gradient algorithm. The starting estimate for the new mesh velocity vector may be extrapolated from previous timesteps, e.g.

$$\mathbf{w}_0^{n+1} = 2\mathbf{w}^n - \mathbf{w}^{n-1}. \tag{12.20}$$

As the mesh movement is linked to a timestepping algorithm for the fluid part, and the body movement occurs at a slow pace compared to the other wavespeeds in the coupled fluid/solid system, normally no more than two to five steps are required to smooth the velocity field sufficiently, i.e. $3 \leq n \leq 5$. The overhead incurred by this type of smoothing is very small compared to the overall costs for any ALE-type methodology for Euler solvers. The performance of iterative solvers can deteriorate considerably for highly anisotropic grids, such as those commonly employed by RANS solvers. For such cases, it is convenient to use linelets (Soto *et al.* (2003)) or block preconditioners, and to obtain a better prediction of the mesh velocity at the next timestep.

In what follows, the *basic assumption* is that **K** does not change significantly in time. For many situations this is indeed the case. The mesh does not move significantly from timestep to timestep, and the distance from the bodies for individual gridpoints also does not change considerably.

If we denote by $\mathbf{w}^i, \mathbf{r}^i, i = 1, l$ the values of the mesh velocity and RHSs at previous timesteps $n - i$, we know that

$$\mathbf{K} \cdot \mathbf{w}^i = \mathbf{r}^i. \tag{12.21}$$

Given the new RHS **r**, we can perform a least-squares approximation to it in the basis $\mathbf{r}^i$, $i = 1, l$,

$$(\mathbf{r} - \alpha_i \mathbf{r}^i)^2 \rightarrow \min, \tag{12.22}$$

which results in

$$\mathbf{A}\boldsymbol{\alpha} = \mathbf{s}, \quad A^{ij} = \mathbf{r}^i \cdot \mathbf{r}^j, \quad s^i = \mathbf{r}^i \cdot \mathbf{r}. \tag{12.23}$$

Having solved for the approximation coefficients α_i, we can estimate the start value **w** from

$$\mathbf{w} = \alpha_i \mathbf{w}^i. \tag{12.24}$$

Note that, in principle, the use of the RHSs $\mathbf{r}^i, i = 1, l$ as a basis may be numerically dangerous. After all, if any of these vectors are parallel, the matrix **A** is singular. One could perform a Gram–Schmidt orthogonalization instead (Fischer (1998)). However, this has not been found to be a problem. The advantage of using simply the original RHSs is that the update of the basis is straightforward. One keeps an index for the last entry in the basis, and simply inserts the new entries at the end of the timestep in the position of the oldest basis vector. The storage requirements for this projective predictor scheme are rather modest: `2*ndimn*npoin*nvecl`. Typically one to four basis vectors are used, i.e. the storage is at most `24*npoin`. The effect of using even two search directions is dramatic: for RANS grids the number of mesh velocity iterations per timestep drops from $O(20)$ to $O(2)$.

If the diffusion coefficient appearing in (12.17) is set to $k = 1$, a true Laplacian velocity smoothing is obtained. This yields the most ‘uniform deformation’ of elements, and therefore minimizes the number of re-meshings or re-maps required. Alternatively, for element-based codes, one may approximate the Laplacian coefficients K^{ij} in (12.18) by

$$\nabla^2 \mathbf{w} \approx -(\mathbf{M}_l - \mathbf{M}_c) \cdot \mathbf{w}. \tag{12.25}$$

This approximation is considerably faster for element-based codes (for edge-based codes there is no difference in speed between the true Laplacian and this expression), but it is equivalent to a diffusion coefficient $k = h^2$. This implies that the gradient of the mesh velocity field will be larger for smaller elements. These will therefore distort at a faster rate than the larger elements. Obviously, for uniform grids this is not a problem. However, for most applications the smallest elements are close to the surfaces that move, prompting many re-meshings.

Based on the previous arguments, one may also consider a diffusion coefficient of the form $k = h^{-p}$, $p > 0$. In this case, the gradient of the mesh velocity field will be larger for the larger elements. The larger elements will therefore distort at a faster rate than the smaller ones, a desirable feature for many applications (see Crumpton and Giles (1995) for an implementation).

12.3.1.1. Variable diffusivity Laplacian smoothing

In most applications, the relevant flow phenomena and associated gradients of density, velocity and pressure are on or close to the bodies immersed in the fluid. Hence, the smallest elements are typically encountered close to the bodies. A straightforward Laplacian smoothing of the mesh velocities will tend to distort the elements in these regions. Thus, the small elements in the most critical regions tend to be the most deformed, leading to a loss in accuracy and possible re-interpolation errors due to the high rate of remeshings required. In an attempt to mitigate this shortcoming, a diffusion coefficient k that is based on the distance d from the moving bodies was proposed by Löhner and Yang (1996). In general, k should be a function of the distance from the body d as sketched in Figure 12.3.

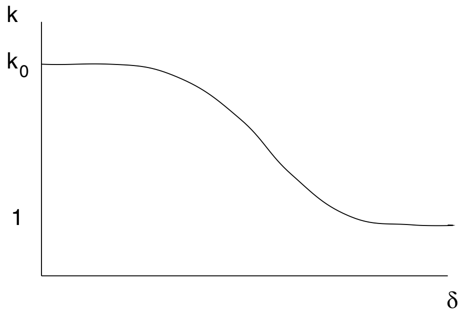


Figure 12.3. Variation of diffusivity as a function of distance

For small d, the 'diffusivity' k should be large, leading to a small gradient of $\mathbf{w}$, i.e. nearly constant mesh velocity close to the moving bodies. For large d, k should tend to unity in order to assure the most uniform deformation of the (larger) elements that are away from the bodies. A shape commonly used in practice is the following constant–ramp–constant function sketched in Figure 12.4:

$$k = k_0 + (1 - k_0)\max\left(0, \min\left(1, \frac{d - d_l}{d_u - d_l}\right)\right). \tag{12.26}$$

The choice of k_0 and the cut-off distances is, in principle, arbitrary. Typical values are $d_l = L/4$, $d_u = L/2$, where L is the 'rigidization distance', and $k_0 = 100$.

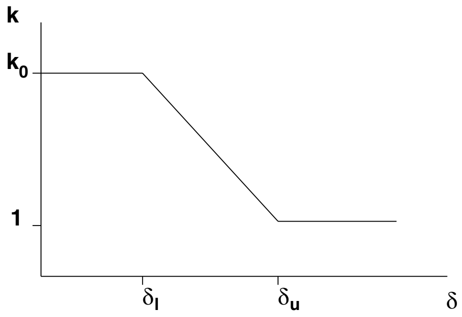


Figure 12.4. Constant–linear–constant function

How to determine efficiently the distance of any given point from a surface has been treated in Chapter 2.

12.3.2. SMOOTHING OF THE COORDINATES

One of the possible disadvantages of mesh velocity smoothing is that the absence of negative elements or inverted volumes cannot be guaranteed. While there are some situations where the appearance of such elements cannot be avoided (e.g. tumbling or rotating objects), a number of authors have preferred to work with the mesh coordinates directly. As before, we start with the prescribed boundary velocities. This yields a new set of boundary coordinates at the new timestep:

$$\mathbf{x}^{n+1}|_\Gamma = \mathbf{x}^n|_\Gamma + \Delta t \cdot \mathbf{w}|_\Gamma. \tag{12.27}$$

Based on these updated values for the coordinates of the boundary points, the mesh is smoothed. Mesh smoothing algorithms have been treated before in Chapter 3. In most cases to date, a simple spring analogy smoother has been employed. Both tension and torsion springs have been employed (see Figure 12.5).

The new values for the coordinates are obtained iteratively via a relaxation or conjugate gradient scheme (Batina (1990a), Rausch *et al.* (1993), Johnson and Tezduyar (1994), Venkatakrishnan and Mavriplis (1995), Johnson and Tezduyar (1997), Farhat *et al.* (1998), Hassan *et al.* (1998), Degand and Farhat (2002), Sorensen *et al.* (2003)). As before, a good initial guess may be extrapolated via

$$\mathbf{x}_0^{n+1} = 2\mathbf{x}^n - \mathbf{x}^{n-1}, \tag{12.28}$$

or via projective prediction of increments (see above). The smoothed mesh velocity is then given by

$$\mathbf{w} = \frac{1}{\Delta t}(\mathbf{x}^{n+1} - \mathbf{x}^n). \tag{12.29}$$

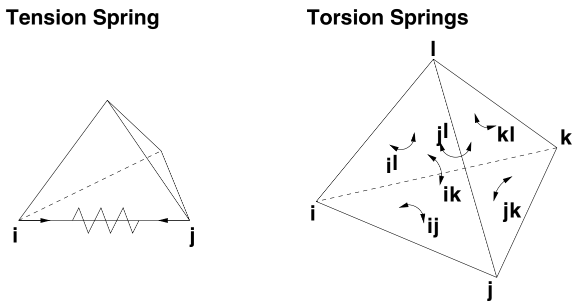


Figure 12.5. Smoothing via springs

Most of the potential problems that can occur for this type of mesh velocity smoothing are due to initial grids that have not been smoothed. For such cases, the velocity of the moving boundaries is superposed on a fictitious mesh smoothing velocity which may be quite large during the initial stages of a run. In order to remedy this situation, one superposes the initial out-of-equilibrium forces to the mesh movement (Venkatakrishnan and Mavriplis (1995)). In this way, if no movement of boundaries occurs, the coordinate positions of points will remain unchanged.

To see more clearly the difference between mesh velocity and coordinate smoothers, consider the simple box shown in Figure 12.6.

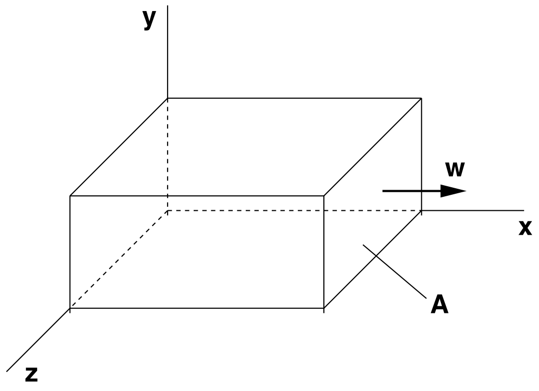


Figure 12.6. Mesh movement: box

Suppose that face A is being moved in the x direction. For the case of mesh velocity smoothing, only displacements in the x direction will appear. This is because the Dirichlet boundary conditions given by (12.12) and (12.13) do not allow any mesh velocity other than that in the x direction to appear. On the other hand, for the case of coordinate smoothing, mesh velocities will, in all likelihood, appear in the y and z directions. This is because, as the mesh moves, the smoothing technique will result in displacements of points in the y and z directions, and hence velocities in the y and z directions.

Instead of springs, which to those more mathematically inclined may seem to be based on merely heuristic constructs, one can also use the equations describing an elastic medium to obtain a smooth displacement field. Denoting by **u** the displacement field, the equations describing an elastic medium undergoing small deformations in the absence of external forces are given by

$$\nabla \cdot \sigma = 0, \sigma = \lambda \operatorname{tr}(\mathbf{E})\mathbf{I} + 2\mu\mathbf{E}, \quad \mathbf{E} = \frac{1}{2}(\nabla\mathbf{u} + {}^{t}\nabla\mathbf{u}), \tag{12.30}$$

which is equivalent to

$$\nabla \cdot \mu\nabla\mathbf{u} + \nabla\lambda(\nabla \cdot \mathbf{u}) + \nabla \cdot (\mu\nabla\mathbf{u}) = 0, \tag{12.31}$$

and for constant λ, μ reduces to

$$\mu\nabla^2\mathbf{u} + (\lambda + \mu)\nabla(\nabla \cdot \mathbf{u}) = 0. \tag{12.32}$$

Comparing (12.31) and (12.32) to (12.17), the similarity of the first term is apparent. The second term in (12.31) and (12.32) is linked to the divergence of the displacement field **u**, which monitors the change in volume of elements. As before, the material parameters μ, λ may be chosen in such a way that close to moving body surfaces the mesh appears more rigid (higher μ, λ), while further away the mesh becomes more flexible (lower μ, λ). Additionally, λ can be made proportional to the volume of elements, in such a way that, as the elements become smaller or negative, λ increases to a large value, making them incompressible. The boundary conditions for the displacement **u** are the same as for the mesh velocity **w** (equations (12.12) and (12.13)). The elasticity equations are best discretized using finite elements (integration by parts, self-adjoint system), but can also be discretized using finite volumes. For iterative solvers, a predictive projection of displacement as described before for mesh velocities yields a considerable reduction in CPU requirements. Mesh movement using the elasticity equations has been used repeatedly (Sackinger *et al.* (1996), Chiandussi *et al.* (2000), Nielsen and Anderson (2001), Soto *et al.* (2002), Stein *et al.* (2004)).

12.3.3. PRESCRIPTION VIA ANALYTIC FUNCTIONS

The third possibility is to prescribe the mesh velocity to be an analytic function based on the distance from the surface. How to determine efficiently the distance of any given point from a surface has been treated in Chapter 2. Given this distance ρ, and the point on the surface closest to it $\mathbf{x}|_\Gamma$, the mesh velocity is given by

$$\mathbf{w} = \mathbf{w}(\mathbf{x}|_\Gamma) f(\rho). \tag{12.33}$$

The function $f(\rho)$ assumes the value of unity for $\rho = 0$, and decays to zero as ρ increases. This makes the procedure somewhat restrictive for general use, particularly if several moving bodies are present in the flowfield. On the other hand, the procedure is extremely fast if the initial distance ρ can be employed for all times (Boschitsch and Quackenbush (1993), Davis and Bendiksen (1993)).

12.4. Region of moving elements

As the elements (or edges) move, their geometric parameters (shape-function derivatives, Jacobians, etc.) need to be recomputed every timestep. If the whole mesh is assumed to be in motion, these geometric parameters need to be recomputed globally. In order to save CPU time, only a small number of elements surrounding the bodies are actually moved. The remainder of the field is then treated in the usual Eulerian frame of reference, avoiding the need to recompute geometric parameters. This may be accomplished in a variety of ways, of which the two most common are:

(a) by identifying several layers of elements surrounding the surfaces that move; and

(b) by moving all elements within a certain distance from the surfaces that move.

These two approaches have their advantages and disadvantages, and are therefore treated in more detail.

(a) *Layers of moving elements*. In this case the elements moved are obtained by starting from the moving surfaces, and performing n passes over nearest-neighbours to construct the n layers of elements that move. This procedure is extremely fast and works only with integer variables. By performing n passes over the mesh, the only data structures required for the construction of the layers are the connectivity matrix `inpoel(1:nnode, 1:nelem)` and a point-array to mark the points that belong to surfaces that move. On the other hand, for situations where the element size varies rapidly the extent of the moving mesh region can assume bizarre shapes. This, in turn, may force many remeshings at a later stage. This type of procedure is most commonly used for Euler calculations (Löhner (1988, 1990), Löhner and Baum (1991), Baum and Löhner (1993), Baum *et al*. (1994, 1995, 1997)).

(b) *Elements within a distance*. This second approach requires the knowledge of the distance of a point from the moving surfaces. How to obtain this information in an optimal way was treated in Chapter 2. All elements within a prescribed distance from the moving surfaces are considered as moving. Although this procedure requires more CPU time when building the necessary data structures, it offers the advantage of a very smooth boundary of the moving mesh region. Moreover, specifying two distances allows the region close to the moving surfaces to be moved in the same way the surfaces move, while further away the mesh velocity is smoothed as before. This allows the movement of grids suitable for RANS simulations. These grids have very elongated elements, and hence are sensitive to any kind of distortion (Martin and Löhner (1992)).

The extent of the region of moving elements chosen, given by the number of layers and/or the distance from the moving surfaces, can have a pronounced effect on the overall cost of a simulation. As the number of elements moved increases, the time interval between regridding increases, but so does the cost per timestep. Therefore, one has to strike a balance between the CPU requirements per timestep and the CPU requirements per regridding. In most of the Euler calculations carried out (Mestreau *et al*. (1993), Baum and Löhner (1993), Baum *et al*. (1994, 1995, 1997, 1998, 1999)) it was found that five to 20 layers of elements represent a good compromise.

12.5. PDE-based distance functions

In situations where objects move slowly through the flowfield it may be advantageous to use a PDE-based distance function instead of recomputing algebraically the distance function. Two common approaches have been used. The first one is based on the Eikonal equation, and requires the complete 'machinery' of hyperbolic conservation laws for a satisfactory solution. The second is an approximate distance function based on the solution of a Laplacian. It is far simpler to implement, but will yield only approximate results.

12.5.1. EIKONAL EQUATION

Denoting the distance to the surface Γ as $d(\mathbf{x})$, it is clear that this function has to satisfy

$$|\nabla d| = 1, \quad d_\Gamma = 0. \tag{12.34}$$

This may be re-written as

$$\frac{\nabla d}{|\nabla d|} \cdot \nabla d = 1. \tag{12.35}$$

Furthermore, we may interpret d as the steady-state result of

$$d_{,t} + \mathbf{v} \cdot \nabla d = 1, \quad \mathbf{v} = \frac{\nabla d}{|\nabla d|}, \tag{12.36}$$

which is a nonlinear hyperpolic equation with the advection velocity given by the unit gradient of the distance $d(\mathbf{x})$. Writing the original system given by (12.34) in this way has the advantage that the eigenvalue of the system is always unity. In order to solve this equation, one can proceed in any of the ways outlined before for advection-dominated flows: Taylor–Galerkin, Streamline-Upwind Petrov-Galerkin (SUPG), FCT, upwind TVD solvers, etc. If the mesh movement is such that the distance function d at each point changes only slowly, the convergence of (12.36) to a reasonable steady state is very quick, requiring three to five passes over the mesh.

12.5.2. LAPLACE EQUATION

Consider the 1-D Poisson problem:

$$d_{,xx} = -s; \quad d(0) = 0; \quad d_{,x}|_{x_1} = 0. \tag{12.37}$$

The exact solution is given by

$$d = sx\left(x_1 - \frac{x}{2}\right), \tag{12.38}$$

implying

$$x_1 = \sqrt{\frac{2d_1}{s}}; \quad d_{,x}|_0 = \sqrt{2 \cdot s \cdot d_1}. \tag{12.39}$$

This means that, in order to obtain a unit gradient at $x = 0$, one should choose $s = 1/2d_1$, which in turn leads to $x_1 = 2d_1$. Another way to interpret the results is that the 'distance of interest' x_1 is related to the maximum value of $d = d_1$ and the source strength s.

Another possibility is obtained by noting that for the general case of

$$d_{,xx} = -s; \quad d(0) = 0, \tag{12.40}$$

the exact solution is given by

$$d_{,x} = -sx + c_1; \quad d = -s\frac{x^2}{2} + c_1 x + c_2. \tag{12.41}$$

The Dirichlet condition $d(0) = 0$ implies $c_2 = 0$, and the free constant c_1 can be related to the gradient $d_{,x}$ yielding

$$d = s\frac{x^2}{2} + x d_{,x}, \tag{12.42}$$

from which we can obtain the 'true distance from the wall' given by x:

$$x = \frac{-d_{,x} + \sqrt{(d_{,x})^2 + 2d}}{s}. \tag{12.43}$$

This simplified analysis is not valid for 2-D and 3-D solutions of the general Poisson problem

$$\nabla^2 d = -s; \quad d|_{\Gamma_0} = 0; \quad d_{,n}|_{\Gamma_1} = 0, \tag{12.44}$$

where for radial symmetry the solutions contain $\ln(r)$ and $1/r$ terms. On the other hand, for most cases the exact distance from moving bodies is not needed. All that is required is a distance function that will give the proper behaviour for k. The idea is then to use (12.44) to determine the distance function d. The Poisson problem may be solved using iterative procedures, and will converge in two to four iterations for most cases. Moreover, a Possion solver fits naturally into existing CFD codes, where edge-based Laplacian operator modules exist for artificial or viscous dissipation terms. The Neumann condition in (12.44) is enforced by not allowing d to exceed a certain value. After each iterative pass during the solution of (12.44) we impose

$$d \le d_1, \tag{12.45}$$

which in effect produces the desired Neumann boundary condition at Γ_1. The generalization of (12.43) for $s = 1$ yields

$$d_\xi = -|\nabla d| + \sqrt{|\nabla d|^2 + 2d}, \tag{12.46}$$

where d_ξ denotes the closest distance to the wall.

12.6. Penalization of deformed elements

An observation often made is that, as a mesh deforms, the appearance of distorted (or even inverted) elements is not uniform across regions, but happens to individual elements. In order to delay the required removal, remeshing or diagonal switching of these regions, one can attempt to 'stiffen' or 'penalize' these elements. Recall that, for the Laplacian smoothing of velocities, a high value of the stiffness k in (12.17) implies that the element will move rigidly. Many measures of quality may be employed (see Chapter 3). A convenient one is given by the following expression:

$$Q = \frac{h_{\max} S}{V}, \tag{12.47}$$

where $h_{\max}$, S and V denote the maximum edge length, total surface area and volume of a tetrahedron, respectively. Observe that as an element distorts into a sliver the volume will approach zero, while $h_{\max}$, S change very little. The element stiffness due to distortion is then given by $k = k(Q)$. A typical expression often employed is

$$k = \min(k_{\max}, c_k Q/Q_{\mathrm{ideal}}), \tag{12.48}$$

there $k_{\max}$, c_k are constants. The final stiffness user for mesh displacement or mesh velocity smoothing is then

$$k = \max(k(\delta), k(Q)). \tag{12.49}$$

12.7. Special movement techniques for RANS grids

The arbitrary movement of grids suitable for RANS simulations, i.e. with aspect ratios in excess of 1:100, can present formidable difficulties. Even minor differences in mesh velocity can lead to negative elements in the close-wall region. Consider the end of a beam structure surrounded by a RANS grid as shown in Figure 12.7. On the far right, the mesh velocity will vanish. The beam, as it oscillates in the vertical direction, will induce a shearing of the mesh. Even for high stiffness coefficients k, a Laplacian smoothing may be insufficient in avoiding mesh velocity differences that, in time, create negative elements at the corners.

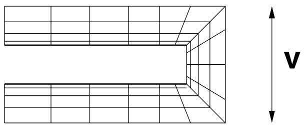


Figure 12.7. Deforming RANS mesh

The best way to treat the movement of highly stretched grids in the near-wall regions is by prescribing the displacement/mesh velocity in layers (Nielsen and Anderson (2001), Soto and Löhner (2002), Sorensen *et al.* (2003), Soto *et al.* (2004)). The number of layers may be chosen according to a prescribed distance, a prescribed number of layers or when a threshold of grid anisotropy has been reached.

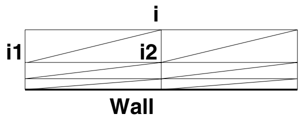


Figure 12.8. Deforming RANS mesh

The displacement/mesh velocity of the next layer is obtained from a weighted average of all nearest-neighbours in the layer below (see Figure 12.8):

$$\mathbf{v}_i = \sum_{ij,l(j)<l(i)} w_{ij}\mathbf{v}_j. \tag{12.50}$$

In order to favour the nearest-neighbours in the stretched grids, the weights are made to be proportional to the inverse of the distances:

$$c_{ij} = \frac{1}{|\mathbf{x}_i - \mathbf{x}_j|}, \quad w_{ij} = \frac{c_{ij}}{\sum_{ik,l(k)<l(i)} c_{ik}}. \tag{12.51}$$

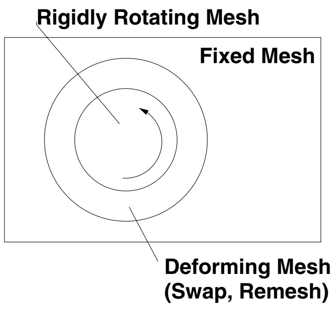


Figure 12.9. Ring approach for rotating machinery

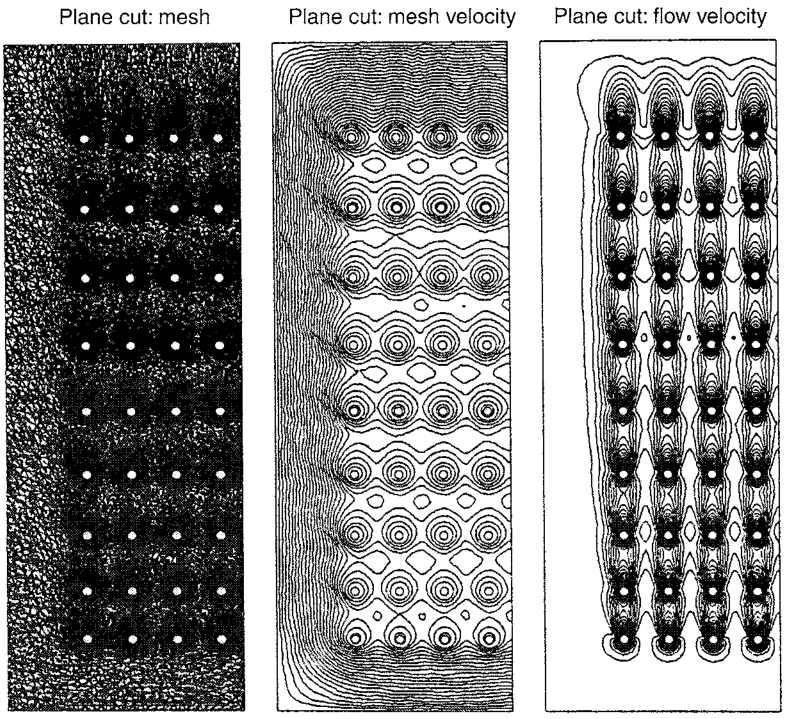


Figure 12.10. Multiple spheres falling towards a wall

12.8. Rotating parts/domains

Many engineering applications involve parts (and the associated domains) that rotate relative to each other. Propellers, turbines and mixers are just a few possible examples. When computing flows for this class of problem using deforming, body conforming grids (as treated in this chapter), several specialized options have been developed. The most common one is the so-called ring approach (Zhu *et al.* (1998)). If we consider the situation sketched in Figure 12.9, the outer grid is not moved. The inner grid rotates rigidly. Only the elements in the ring region between these two are allowed to deform. One can then either remesh this region or simply switch the diagonals. However, simply switching diagonals will not always work, as points that have slightly different radii will tend to overtake each other, potentially creating very small elements. Thus, the option of adding or removing points, which is always possible with local remeshing, should always be available.

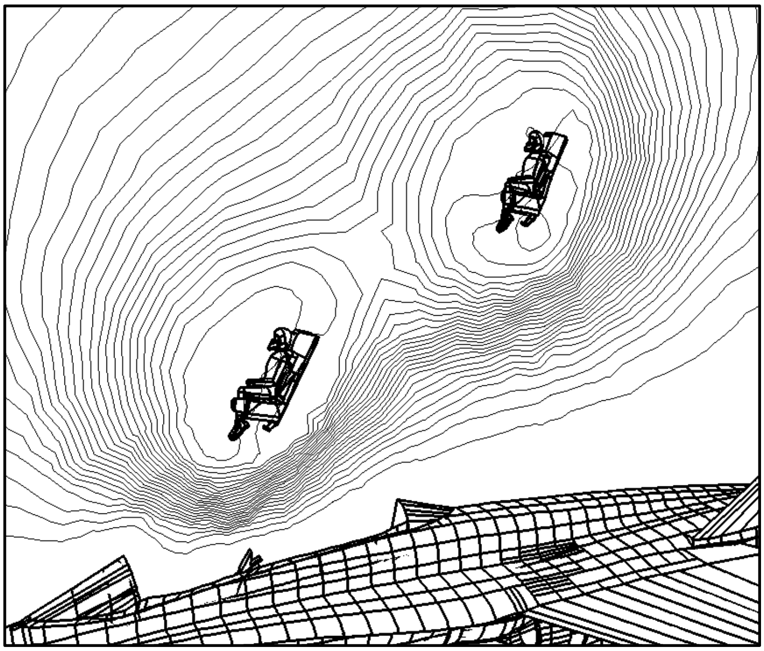


Figure 12.11. Simultaneous seat ejection

12.9. Applications

The following section shows the mesh velocity fields obtained using the techniques described in this chapter for several examples.

12.9.1. MULTIPLE SPHERES

The last decade has seen a continuous interest in the dynamics of bubbles or granular material immersed in a fluid. Figure 12.10 shows a planar cut for a simulation with 36 spheres falling towards the bottom surface. As the spheres get close to the bottom wall, they are removed automatically and the resulting voids are remeshed. Similarly, new spheres are introduced at the top. One can clearly see the velocity field of the mesh movement. Note the almost constant contour levels close to the spheres, which indicates that the mesh in the vicinity of the sphere moves almost rigidly.

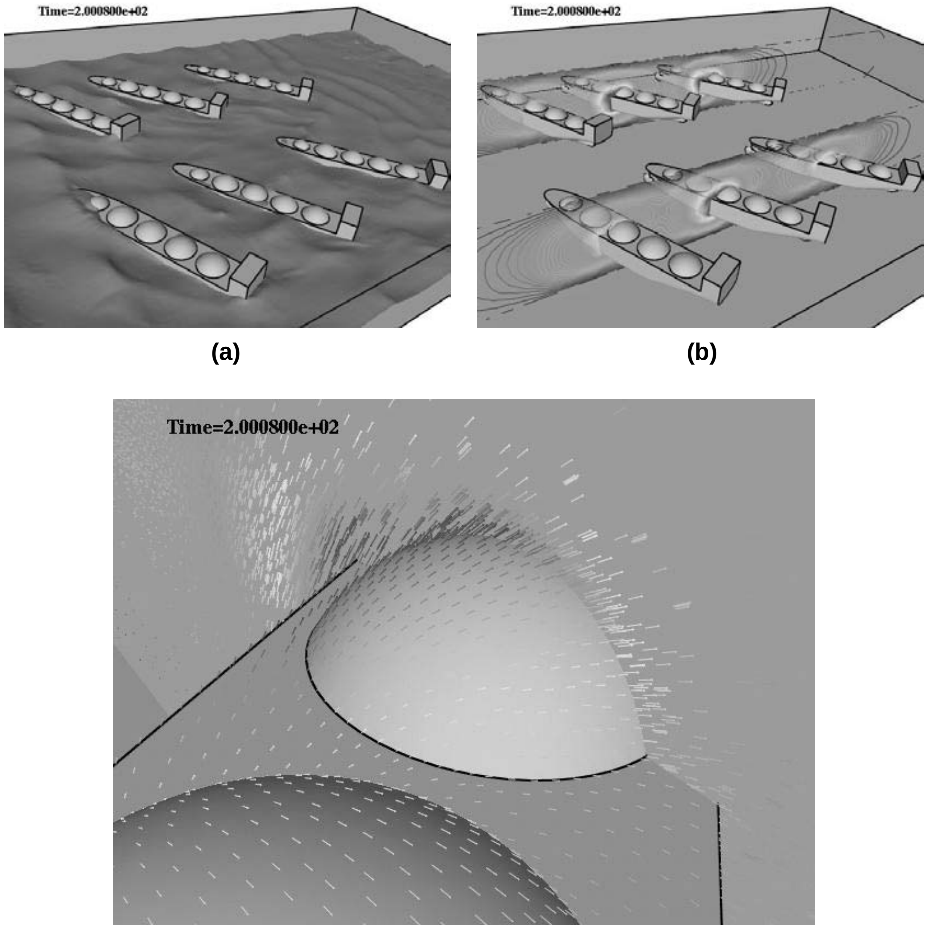


Figure 12.12. Tanker Fleet: (a) free surface; and (b) mesh velocity in two planes; (c) mesh velocity (detail)

12.9.2. PILOT EJECTION FROM F18

Every new generation of ejection seat must be tested for safe deployment. Figure 12.11 shows a planar cut for a simulation with two pilots ejecting simultaneously (Sharov *et al.* (2000b)). As before, the velocity field of the mesh movement displays almost constant contour levels close to the pilots, indicating nearly rigid mesh movement.

12.9.3. DRIFTING FLEET OF SHIPS

This case, taken from Löhner *et al.* (2007c), considers a fleet of tankers that are freely floating objects subject to the hydrodynamic forces of an incoming wavefield. The surface nodes of the ships move according to a 6 degree of freedom integration of the rigid body motion equations, and 30 layers of elements surrounding the ships are moved. The volume mesh has approximately 10 million elements. Figure 12.12(a) shows the free surface at 200 s. The absolute value of the mesh velocity in two planes, as well as a detailed view of the mesh velocity on the surface of one of the tankers and a plane are displayed in Figures 12.12(b) and (c). Several local and global remeshings were required during the run to keep the mesh quality acceptable.

13 INTERPOLATION

The need to interpolate quickly fields of unknowns from one mesh to another is common to many areas of CFD and, more generally, to computational mechanics and computational physics. The following classes of problems require fast interpolation algorithms.

(a) *Simulations that require different grids as the solution proceeds*. Examples of this kind are adaptive remeshing for steady-state and transient simulations (Löhner and Ambrosiano (1990), Peraire *et al.* (1992b), Weatherill *et al.* (1993b)), as well as simple remeshing for problems where grid distortion due to movement becomes too severe (Baum and Löhner (1993), Mestreau *et al.* (1993), Löhner *et al.* (1999)).

(b) *Simulations with overlapping grids*. The key idea here is to simplify the mesh generation process and to avoid the regeneration of grids for problems with moving bodies by generating for each component (e.g., wing, pylon, tail, engine inlet, etc.) a local, independent grid. These independent grids overlap in certain regions. The solution to the global problem is obtained by interpolating between the grids after each timestep (Benek *et al.* (1985), Dougherty and Kuan (1989), Meakin and Suhs (1989), Meakin (1993, 1997), Rogers *et al.* (1998), Nakahashi *et al.* (1999)).

(c) *Loose coupling of different codes for multi-disciplinary applications*. In this case, if any of the codes in question are allowed to perform adaptive mesh refinement, the worst-case scenario requires a new interpolation problem at every timestep (Guruswamy and Byun (1993), Rausch *et al.* (1993), Löhner *et al.* (1995, 2004d), Cebral and Löhner (1997)).

(d) *Interpolation of discrete data for the initialization or continuous update of boundary conditions*. Common examples are meteorological simulations, as well as geotechnical data for seepage problems.

(e) *Particle/grid codes*. These codes, commonly used for plasma and particle beam simulations, as well as solid rocket motor applications, require the efficient tracking of a large number ($>10^6$) of particles, making an efficient interpolation algorithm a prime requirement.

(f) *Visualization*. This large class of problems makes extensive use of interpolation algorithms, especially for the comparison of different data sets on similar problems.

In the following, the focus will be on the fast interpolation between different unstructured grids that are composed of the same type of elements, surface-grid-to-surface-grid interpolations and element-to-particle interpolations. In particular, linear triangles and tetrahedra are considered. The ideas developed are general and can be applied to any type of element and grid. On the other hand, other types of grids (e.g. Cartesian structured grids) will lend themselves to specialized algorithms that may be more efficient and easier to implement.

13.1. Basic interpolation algorithm

Consider an unstructured finite element or finite volume mesh, as well as a point p with coordinates $\mathbf{x}_p$. A simple way to determine whether the point p is inside a given element *el* is to determine the shape-function values of p with respect to the coordinates of the points belonging to *el*:

$$\mathbf{x}_p = \sum_i N^i \mathbf{x}_i . \tag{13.1}$$

For triangles in two dimensions and tetrahedra in three dimensions, we have, respectively, two equations for three shape functions and three equations for four shape functions. The sum property of shape functions,

$$\sum_i N^i = 1, \tag{13.2}$$

yields the missing equation, making it possible to evaluate the shape functions from the following system of equations:

$$\begin{Bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{Bmatrix} = \begin{Bmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ z_1 & z_2 & z_3 \\ 1 & 1 & 1 \end{Bmatrix} \cdot \begin{Bmatrix} N^1 \\ N^2 \\ N^3 \\ N^4, \end{Bmatrix} \tag{13.3}$$

or, in concise matrix notation,

$$\mathbf{x}_p = \mathbf{X} \cdot \mathbf{N} \rightarrow \mathbf{N} = \mathbf{X}^{-1} \cdot \mathbf{x}_p . \tag{13.4}$$

Then, the point p is in element *el* if

$$\min(N^i, 1 - N^i) > 0, \quad \forall i. \tag{13.5}$$

For other types of elements more nodes than equations are encountered. The easiest way to determine whether a point is inside an element is to split the element into triangles or tetrahedra and evaluate each of these sub-elements in turn. If the point happens to be in any of them, it is inside the element. This procedure may not be unique for highly deformed bricks, as shown in Figure 13.1. Depending on how the diagonals are taken for the face A–B–C–D, the point to be interpolated may or may not be inside the element. Therefore, subsequent iterations may be required for bricks or higher-order elements with curved boundaries. Other ways to determine whether a point is inside a bilinear element may be found in Westermann (1992).

The criterion set forth in (13.5) is used to determine whether a point lies within the confines of an element. In what follows, different ways of determining the element in question as quickly as possible are considered. The possible improvements in speed depend mostly on the assumptions one can make with respect to the information available.

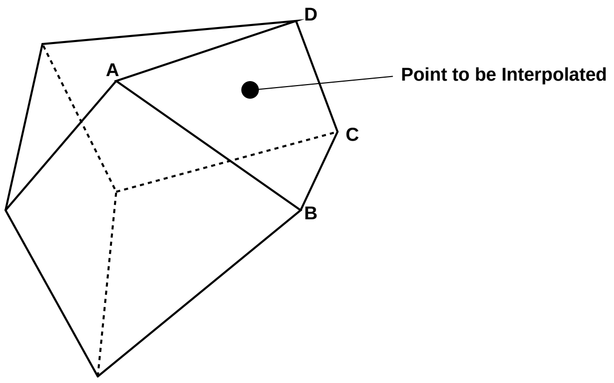


Figure 13.1. Interpolation for a deformed brick element

13.2. Fastest 1-time algorithm: brute force

Suppose we only have a given grid and a point p with coordinates $\mathbf{x}_p$. The simplest way to find the element into which point p falls is to perform a loop over all the elements, evaluating their shape functions with respect to $\mathbf{x}_p$:

```
do ielem=1,nelem                                   ! Loop over all the elements
    Evaluate $N^i$ from Eqn.(13.4);
    if: Criterion (13.5) is satisfied: exit
enddo
```

Since the outer loop over all the elements can readily be vectorized and/or parallelized, this algorithm is extremely fast. This brute force search is used in more refined algorithms both as a start-up procedure and as a fall-back position.

13.3. Fastest *N*-time algorithm: octree search

Suppose that as before we only have a given grid, but instead of just one point p a considerable number of points needs to be interpolated. In this case, the brute-force algorithm described before will possibly require a complete loop over the elements for each point to be interpolated, and, on average, a loop over half the elements. A significant improvement in speed may be realized by only checking the elements that cover the immediate neighbourhood of the point to be interpolated.

A number of ways can be devised to determine the neighbourhood (see Figure 13.2 and Chapter 2):

- bins, i.e. the superposition of a Cartesian mesh (Löhner and Morgan (1987), Meakin (1993));

- octrees, i.e. the superposition of an adaptively refined Cartesian mesh (Knuth (1973), Samet (1984)); and

- alternate digital trees (Bonet and Peraire (1991)).

We consider octrees here, as bins perform poorly for problems where the nearest-neighbour distances vary by more than two orders of magnitude in the domain. One may form an octree with the element centroids or points. It is advisable to choose the latter option, as for 3-D tetrahedral grids the number of points is significantly less than the number of elements. Denoting by `coint(1:ndimn,1:npint)` the coordinates of the `npint` points to be interpolated, the octree search algorithm proceeds as follows:

```
For the given mesh:
    Form the octree for the points
    Form the list of elements surrounding points (esup1,esup2)

do ipint=1,npint                              ! Loop over the points to be interpolated
    Obtain close points of given mesh from the octree;
    Obtain the elements surrounding the close points from esup1,esup2, and store in
    lcloe(1:ncloe)
       do icloe=1,ncloe                             ! Loop over the close elements
          ielem=lcloe(icloe)
           Evaluate $N^i$ from equation (13.4);
          if: Criterion (13.5) is satisfied: exit
       enddo
    As we have failed to find the host element: Use brute-force
enddo
```

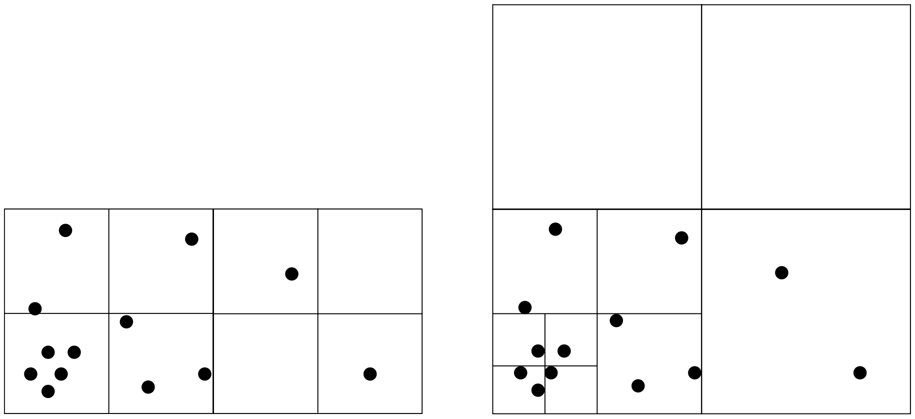


Figure 13.2. Algorithms to determine spatial proximity: (a) bin; (b) quadtree

Several improvements are possible for this algorithm. One may, in a first pass, evaluate the point of the given mesh closest to $\mathbf{x}_p$, and only consider the elements surrounding that point. In general this pass is successful. Should it fail, the elements surrounding all the close points are considered in a second pass. If this second pass also fails (see Figure 13.3 for some pathological cases), one may either enlarge the search region, or use the brute-force algorithm described above. The octree search algorithm is scalar for the first (integer) phase

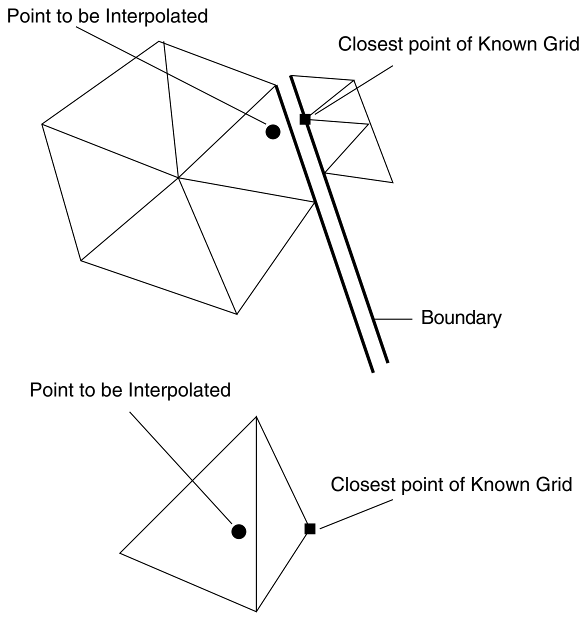


Figure 13.3. Interpolation with internal boundaries

(obtaining the close points and elements), but all other stages may be vectorized. The vector lengths obtained for 3-D grids are generally between 12 and 50, i.e. sufficiently long for good performance.

13.4. Fastest known vicinity algorithm: neighbour-to-neighbour

Suppose that as before we only have a given grid and a considerable number of points need to be interpolated. Moreover, assume that, for any given point to be interpolated, an element of the known grid that is in the vicinity is known. In this case, it may be faster to jump from neighbour to neighbour in the known grid, evaluating the shape-function criterion given by (13.5) (see Figure 13.4).

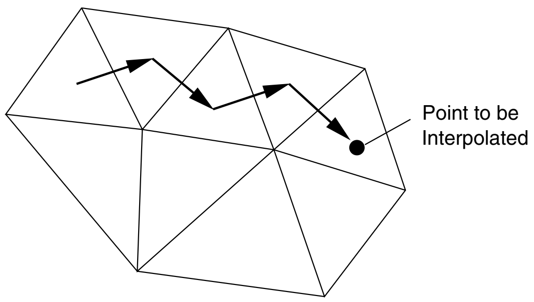


Figure 13.4. Neighbour-to-neighbour search

If the element into which **x** falls can be found in a few attempts (<10), this procedure, although scalar, will outperform all other ones. The neighbour-to-neighbour search algorithm may be summarized as follows:

```
For the given mesh:
   Form the list of elements adjacent to elements
   ⇒  esuel(1:nfael,1:nelem)

do ipint=1,npint                         ! Loop over the points to be interpolated
   iesta=lesta(ipint)                    ! Obtain a good starting element
   do itry=1,mtry                        ! Loop over the nearest neighbour tries
      For iesta: Evaluate $N^i$ from equation (13.4);
      if (13.5) is satisfied then
         exit
      else
         Obtain node inmin for min($N^i$)
         iesta=esuel(inmin,iesta)        ! Jump to neighbour
      endif
   enddo
enddo
```

The neighbour-to-neighbour algorithm performs very well in the domain, but can have problems on the boundary. Whereas the brute-force and octree search algorithms can ‘jump’ over internal or external boundaries, the neighbour-to-neighbour algorithm can stop there (see Figure 13.5). Its performance depends heavily on how good a guess the starting element `iesta` is: it can be provided by bins, octrees or alternate digital trees. On the other hand, due to its scalar nature, such an algorithm will not be able to compete with the octree search algorithm described in Chapter 3. Its main use is for point-to-grid or grid-to-grid transfer, where a very good guess for `iesta` may be provided. This fastest grid-to-grid interpolation technique is described in the next section.

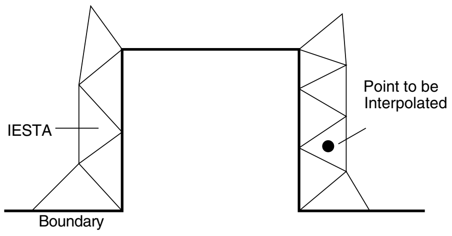


Figure 13.5. Internal boundaries

13.5. Fastest grid-to-grid algorithm: advancing-front vicinity

The crucial new assumption made here as opposed to all the interpolation algorithms described so far is that the points to be interpolated belong to a grid, and that the grid connectivity (e.g. the points belonging to each element) is given as input. In this case, whenever the element `ieend` of the known grid into which a point of the unknown grid falls is found, all the surrounding points of the unknown grid that have not yet been interpolated are given as a starting guess `ieend`, and stored in a list of ‘front’ points `lfrnt`. The next point to be interpolated is then drawn from this list, and the procedure is repeated until all points have been interpolated. The procedure is sketched in Figure 13.6, where the notion of ‘front’ becomes apparent.

The complete algorithm may be summarized as follows:

```
For the given mesh:
  Form the list of elements adjacent to elements
  ⇒ esuel(1:nfael,1:nelem)
For the unknown mesh:
  Form the list of points surrounding points ⇒ psup1, psup2
lpunk(1:npunk)=0                       ! Mark points of unknown grid as untouched
nfrnt=1                  ! Initialize list of front points lfrnt for unknown grid
lfrnt(1)=1
lpunk(1)=1
while(nfrnt.gt.0):
   nfrn1   =0                                              ! Initialize next front
   do ifrnt=1,nfrnt                                       ! Loop over front points
      ipunk=lfrnt(ifrnt)                                  ! Point of unknown grid
      iesta=lpoi2(ipunk)                          ! Starting element in known grid
Attempt nearest neighbour search for ntry attempts
      if Unsuccessful: Use brute force
         if Unsuccessful: Stop or skip
         else
            ⇒ ieend
         endif
      endif
Store shape-functions and host element ieend
      lpunk(ipunk)=ieend                             ! Mark point as interpolated
! Loop over points surrounding ipunk
      do istor=psup2(ipunk)+1,psup2(ipunk+1)
         jpunk=psup1(istor)
         if(lpunk(jpunk).eq.0) then
Store ieend as starting element for this point
            lpunk(jpunk)=ieend
            nfrn1=nfrn1+1                              ! Update new front counter
            lfrn1(nfrn1)=jpunk                 ! Include this point in the new front
         endif
      enddo
  enddo
Reset nfrnt, and see if we have a new front
   nfrnt=nfrn1
   if(nfrnt.gt.0) then
Transcribe the new front and proceed
      lfrnt(1:nfrnt)=lfrn1(1:nfrnt)
   endif
endwhile
```

Several possible improvements for this algorithm, layering of brute-force searches, inside-out interpolation, measuring concavity and vectorization, are detailed in the following.

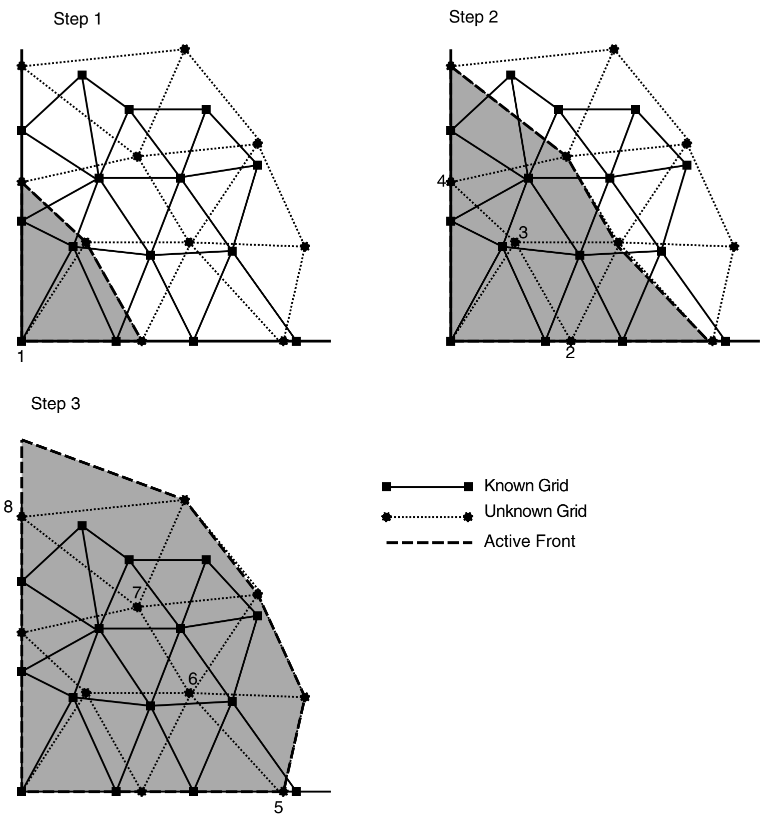


Figure 13.6. Advancing-front vicinity search

13.5.1. LAYERING OF BRUTE-FORCE SEARCHES

In most instances (the exception being grids with very large disparity in element size where `ntry` attempts are not sufficient), the neighbour-to-neighbour search will only fail on the boundary. Therefore, a first brute-force search over the boundary elements of the known grid will reduce brute-force search times considerably. Note, however, that one has to know the boundary points in this case. The elements of the known grid are renumbered so that all elements with three or more nodes on the boundary in three dimensions and two or more nodes on the boundary in two dimensions appear at the top of the element list. These `nelbo` $<$ `nelem` elements are scanned first whenever a brute-force search is required. Moreover, after a front has been formed, only these elements close to boundaries are examined whenever a brute-force search is required.

13.5.2. INSIDE-OUT INTERPOLATION

This improvement is directed to complex boundary cases. Under this category we group cases where the boundary has sharp concave corners or ridges, or those cases where, due to the concavity of the surface, points may be close but outside of the known grid (see Figure 13.7).

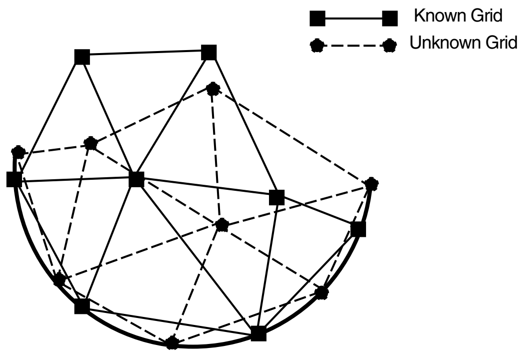


Figure 13.7. Problems at concave boundaries

In this latter case, it is advisable to form two front lists, one for the interior points and one for the boundary points. The interpolation of all the interior points is attempted first, and only then are the boundary points interpolated. This procedure drastically reduces the number of brute-force searches required for the complex boundary cases listed above. This may be seen from Figure 13.8, where the brute-force search at the corner was avoided by this procedure. As before, knowledge of the boundary points is required for this improvement.

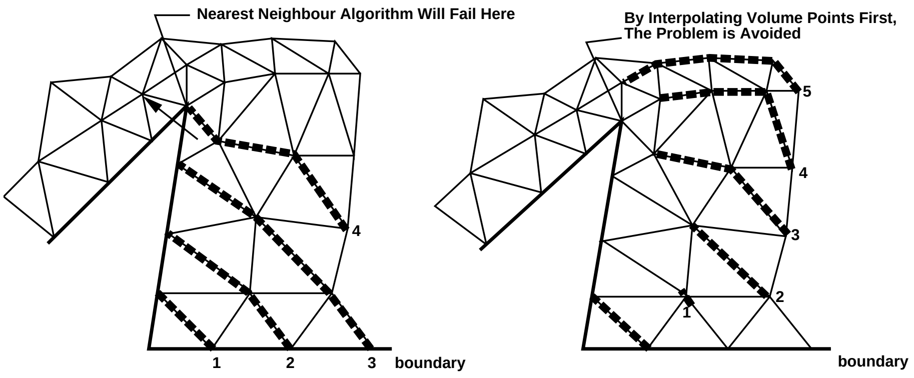

Nearest Neighbour Algorithm Will Fail Here
4
1
2
3
boundary
By Interpolating Volume Points First, The Problem is Avoided
5
4
3
2
1
boundary

Figure 13.8. Inside-out interpolation

13.5.3. MEASURING CONCAVITY

For concave surfaces, criterion (13.5) will not be satisfied for a large number of surface points, prompting many brute-force searches. The algorithmic complexity of the interpolation procedure could potentially degrade to $O(N_b^2)$, where N_b is the number of boundary points.

The number of brute-force searches can be reduced considerably if the concavity of the surface can be measured. Assuming the unit face-normals **n** to be directed away from the domain, a possible measure of concavity is the visibility of neighbouring faces from any given face. With the notation of Figure 13.9, the concavity of a region along the boundary may be determined by measuring the normal distance between the face and the centroids of the neighbouring faces. The allowable distance from the face for points to be interpolated is then given by some fraction α of the minimum distance measured:

$$d = \alpha |\min(0, \mathbf{n} \cdot (\mathbf{x}_0 - \mathbf{x}_i))|. \tag{13.6}$$

Typical values for α are $0.5 < \alpha < 1.5$. If a neighbour-to-neighbour search ends with a boundary face (`esuel(inmin,iesta)=0`), and all other shape functions except the one corresponding to `inmin` satisfy (13.5), the distance of the point to be interpolated from the face is evaluated. If this distance is smaller than the one given by (13.6), the point is accepted and interpolated from the current element `iesta`. Otherwise, a brute force search is conducted.

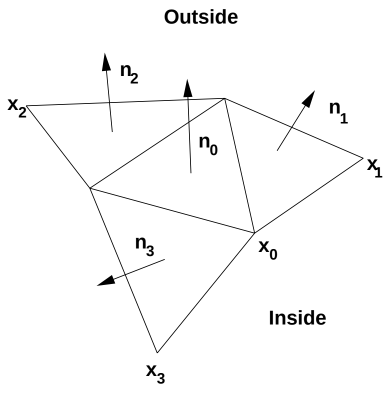


Figure 13.9. Measuring concavity

The application of this procedure entails an element of risk, particularly if two concave boundaries are very close together or overlapping (see Figure 13.10). Moreover, a number of additional arrays are required: face arrays, a distance array to store the concavity, and the relation between element faces and the face array. This last relation can easily be obtained by setting the negative value of the face number whenever `esuel(inmin,ielem)=0`.

13.5.4. VECTORIZATION

The third possible improvement is vectorization. The idea is to search for all the points on the active front at the same time. It is not difficult to see that, for large 3-D grids, the vector lengths obtained by operating in this manner are considerable, leading to very good overall performance. To obtain a vectorized algorithm we must perform the steps described above in vector mode, executing the same operations on as many uninterpolated points as possible. The obstacle to this approach is that not every point will satisfy criterion (13.5) in the same number of attempts or passes over the points to be interpolated. The solution is to reorder the

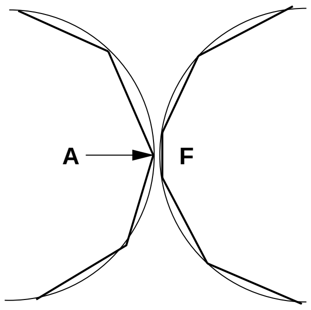


Figure 13.10. Close concave boundaries

points to be interpolated after each pass so that all points that have as yet not found their host element are at the top of the list. Such an algorithm proceeds as follows:

```
nprem=nfrnt             ! Initialize the remaining number of points to be interpolated
while(nprem.gt.0):
   do iprem=1,nprem                        ! Vector loop over the remaining points
      ipunk=lprem(iprem)                             ! Point to be interpolated
       Evaluate (13.3-13.5)
   enddo
   npnxt=0                                      ! Sort out the remaining points
   nptra=0
   do iprem=1,nprem
      if: Criterion (13.5) satisfied then
          nptra=nptra+1
          lptra(nptra)=lprem(iprem)
      else
          npnxt=npnxt+1
          lpnxt(npnxt)=lprem(iprem)
      endif
   enddo
   if(npnxt.eq.0) then                    ! See if all points have been interpolated
      exit
   else
      lpcur(1:npnxt)=lpnxt(1:npnxt)                        ! Reorder the points
      lpcur(npnxt+1,npcur)=lptra(1:nptra)
      nprem=npnxt                                           ! Reset nprem
   endif
endwhile
```

One can reduce the additional memory requirements associated with indirect addressing by breaking up all loops over the `nprem` remaining points into subgroups. This is accomplished automatically by using scalar temporaries on register-to-register machines. For memory-to-memory machines, a maximum group vector length must be specified by the user.

In order to illustrate the gains in speed through vectorization, consider the interpolation between different grids for the volume inside a cube or around a train configuration.

Figures 13.11 and 13.12 show the surface grids of some of these grids. Table 13.1 summarizes the performance recorded on the Cray-C90 for the advancing-front vicinity algorithm. BFS denotes the number of brute force searches required. Both the scalar and vector version of the algorithm were carefully optimized for speed.

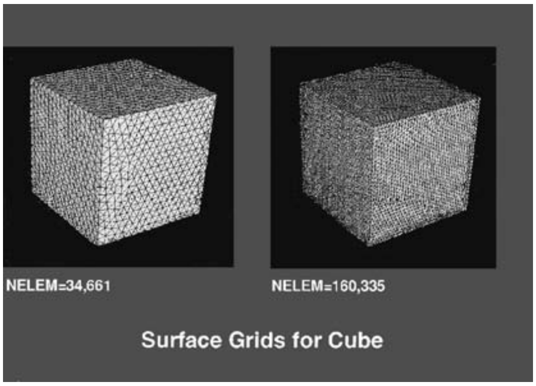


Figure 13.11. Cube

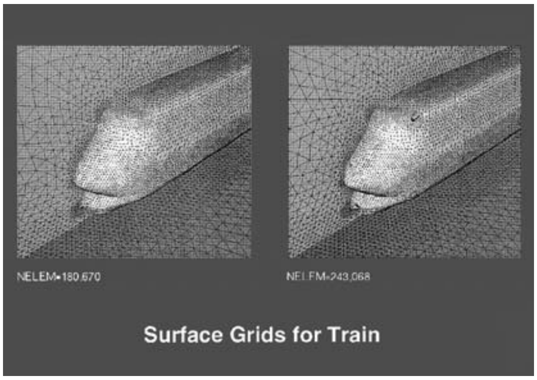


Figure 13.12. Generic train

One can observe speedups between 1:3.3 and 1:5.0 for the vectorized version. The interpolation speed per point per full interpolation varied between 3.7×10^{-6} s/point and 7.4×10^{-6} s/point. This number, as well as the speedup obtained, depends on the number of brute-force interpolations required, as well as the average number of tries required in order to find the host element for each point to be interpolated. The more tries required, the higher the speedup achieved by the vectorized version, as the transcription and rearrangement costs are amortized over more vectorized CPU-intensive operations. For the third and fourth cases, the only difference is the number of brute-force interpolations required. The train configuration, which is typical of CFD runs with moving bodies that require many re-interpolations of the solution during a run (Mestreau *et al.* (1993)), had a few concave surfaces. Some of the points of the second grid were outside the first mesh, prompting a search over all elements

Table 13.1. Mesh-to-mesh interpolation timings

Case	`nelem`$_1$	`nelem`$_2$	# BFS	Scalar	Vector	Speedup
Cube$_1$	34 661	30 801	0	0.1399	0.0283	4.94
Cube$_2$	34 661	160 355	0	0.5360	0.1104	4.86
Train$_1$	180 670	243 068	31	1.1290	0.3405	3.32
Train$_2$	180 670	243 068	0	0.9905	0.2020	4.90

with three or more nodes on the boundary. As one can see, this exhaustive search, which runs at 325 Mflops on the Cray-C90, does not have any significant affect on the performance of the scalar version. The corresponding timings for the vectorized version, however, deteriorate significantly. We note that the number of elements with three or more nodes on the boundary only constitutes about 9% of the total number of elements. Doing an exhaustive search over the complete mesh would therefore have increased interpolation times dramatically.

13.6. Conservative interpolation

The interpolation schemes described so far will attempt to satisfy, in a pointwise manner, for grids 1,2,

$$u_2(\mathbf{x}) \approx u_1(\mathbf{x}). \tag{13.7}$$

Besides this local accuracy, in many cases the satisfaction of a global conservation constraint of the form

$$\int_\Omega u_2 \, d\Omega = \int_\Omega u_1 \, d\Omega \tag{13.8}$$

is required. Typical examples are the conservation of mass for compressible flow problems, where

$$\int_\Omega \rho_2 \, d\Omega = \int_\Omega \rho_1 \, d\Omega, \tag{13.9}$$

or the conservation of forces for fluid-structure interaction problems:

$$\int_\Gamma \mathbf{s}_2 \, d\Gamma = \int_\Gamma \mathbf{s}_1 \, d\Gamma. \tag{13.10}$$

Writing the unknowns on both grids as

$$u_1 = N_1^i \hat{u}_{i1}, \quad u_2 = N_2^i \hat{u}_{i2}, \tag{13.11}$$

let us consider the weighted residual statement (or L2 projection)

$$\int N_2^i N_2^j \, d\Omega \, \hat{u}_{j2} = \int N_2^i N_1^j \, d\Omega \, \hat{u}_{j1}. \tag{13.12}$$

This may be expressed in matrix form as

$$\mathbf{M}_c \mathbf{u}_2 = \mathbf{r} = \mathbf{L}\mathbf{u}_1, \tag{13.13}$$

where $\mathbf{M}_c$ is the familiar consistent mass matrix, and $\mathbf{L}$ a projection matrix. As the shape functions satisfy the interpolation property ($\sum_i N_s^i(\mathbf{x}) = 1$), we can express the conservation statement given by (13.8) as

$$\int u_2\, d\Omega = \int N_2^j\, d\Omega\, \hat{u}_{j2} = \int \sum_i N_2^i N_2^j\, d\Omega\, \hat{u}_{j2} = \sum_i \int N_2^i N_2^j\, d\Omega\, \hat{u}_{j2}$$
$$= \sum_i \int N_2^i N_1^j\, d\Omega\, \hat{u}_{j1} = \int \sum_i N_2^i N_1^j\, d\Omega\, \hat{u}_{j1} = \int N_1^j\, d\Omega\, \hat{u}_{j1} = \int u_1\, d\Omega. \tag{13.14}$$

The weighted residual statement given by (13.12) therefore leads to a conservative transfer of unknowns as expressed by (13.8). However, one is faced with the integrals of shape functions of different grids required to evaluate the projection matrix

$$\mathbf{r} = \mathbf{L}\mathbf{u}_1 = \int N_2^i N_1^j\, d\Omega\, \hat{u}_{j1}. \tag{13.15}$$

An option often advocated for 1-D and 2-D cases is to compute all the intersections of elements, retriangulate and then evaluate the resulting integrals exactly. This works well for linear elements in one and two dimensions, but becomes onerous in three dimensions and nearly impossible for curved isoparametric elements of higher order in three dimensions. A recourse that has worked well for many problems is to evaluate the integrals in (13.15) via numerical quadrature. Consider first the option of performing a loop over the elements of mesh 2 (the mesh to which the solution is being interpolated). The residual vector r^i for every point of mesh 2 is evaluated as

$$r^i = \int N_2^i N_1^j\, d\Omega\, \hat{u}_{j1} = \sum_2 V_2 \sum_{qp} W_{qp} N_2^i(\mathbf{x}_{qp}) N_1^j(\mathbf{x}_{qp}) \hat{u}_{j1}. \tag{13.16}$$

Here V denotes the volumes of the elements and W_{qp}, $\mathbf{x}_{qp}$ the weights and locations of the (Gaussian) quadrature points. In this case the shape-function values at the quadrature points on mesh 2 ($N_2^i(\mathbf{x}_{qp})$) are known, whereas the unknowns of mesh 1 at the quadrature points ($u_1(\mathbf{x}_{qp})$) have to be obtained via interpolation. The procedure is shown diagrammatically for a 1-D case in Figure 13.13. Note that even though every point of mesh 2 has a defined interpolated value from mesh 1, the procedure is not guaranteed to be conservative: if the element size of mesh 1 is smaller than the element size of mesh 2 $h_1 < h_2$ some points of mesh 1 may not contribute to mesh 2.

Consider next the option of performing a loop over the elements of mesh 1 (the mesh from which the solution is being interpolated). The residual vector r^i for every point of mesh 2 is evaluated as

$$r^i = \int N_2^i N_1^j\, d\Omega\, \hat{u}_{j1} = \sum_1 V_1 \sum_{qp} W_{qp} N_2^i(\mathbf{x}_{qp}) N_1^j(\mathbf{x}_{qp}) \hat{u}_{j1}. \tag{13.17}$$

In contrast to the previous case, now the unknowns of mesh 1 at the quadrature points ($u_1(\mathbf{x}_{qp})$) are known, whereas the shape-function values at the quadrature points on mesh 2 ($N_2^i(\mathbf{x}_{qp})$) have to be obtained/interpolated. The procedure is shown diagrammatically for

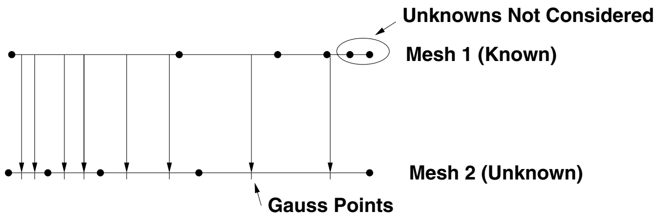


Figure 13.13. Projection via quadrature: loop over elements of mesh 2

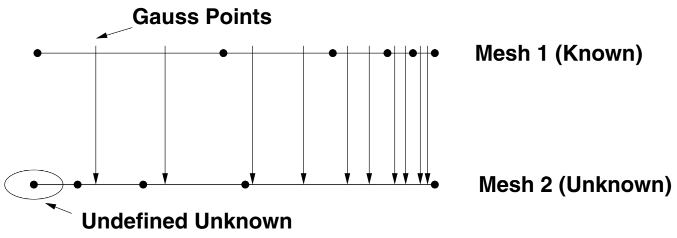


Figure 13.14. Projection via quadrature: loop over elements of mesh 1

a 1-D case in Figure 13.14. Note that even though every point of mesh 1 has been used, thus guaranteeing conservation, the procedure may not be pointwise accurate. Indeed, if the element size of mesh 2 is smaller than the element size of mesh 1 $h_2 < h_1$ some points of mesh 2 may not have any interpolated value.

The recourse taken is to employ an *adaptive Gaussian quadrature* as proposed by Cebral (1997). In a first pass, the element size between the grids is interpolated and compared. Depending on the size ratio of elements, the number of Gauss points is increased to guarantee a conservative and accurate transfer of information (see Figure 13.15). Given that the main reason to add Gauss points per element is not the order of polynomial appearing in the integrals, but rather the discrepancy in element size, it is advisable to use the same Gaussian quadrature throughout the mesh, and to 'subdivide' the elements where more quadrature points are required. From an algorithmic perspective, the use of adaptive Gaussian quadrature does not add any further complexity vis-à-vis standard interpolation. What changes is the information required: whereas for standard interpolation one requires the host element of mesh 1 for every point of mesh 2, for conservative projection one requires the host element of mesh 2 for every quadrature point of mesh 1. Typically, one first obtains the host element of mesh 2 for every point of mesh 1, and then uses these as starting elements in a near-neighbour search when looping over the quadrature points.

13.6.1. CONSERVATIVE AND MONOTONIC INTERPOLATION

Like any higher-order interpolation, the weighted residual projection given by (13.13) can lead to loss of monotonicity. This may be tolerable in most situations, but in some it may prevent the continuation of a run (just imagine what a negative density can do to the speed of sound, or an overshoot in temperature to an igniting fluid). Given that (13.13) is conveniently

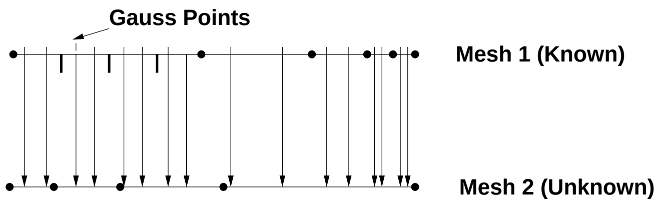


Figure 13.15. Projection via quadrature: adaptive loop over elements of mesh 1

solved iteratively as

$$\mathbf{M}_l \mathbf{u}_2^{i+1} = \mathbf{r} + (\mathbf{M}_l - \mathbf{M}_c)\mathbf{u}_2^i, \quad i = 1, niter, \tag{13.18}$$

with $\mathbf{u}^0 = 0$ and $\mathbf{M}_l$ the lumped mass matrix, one can borrow the concept of flux-corrected transport (FCT, see Chapter 9) and interpret the solution obtained after the first pass,

$$\mathbf{M}_l \mathbf{u}_2^l = \mathbf{r}, \tag{13.19}$$

as a 'low-order solution'. Indeed, as there is no near-neighbour coupling via the left-hand side (consistent mass matrix), the solution tends to be of lower accuracy and present almost no under/overshoots. One can arrive at a conservative and monotonic projection by following the basic FCT steps:

- compute low-order projection: $\mathbf{M}_l \mathbf{u}^l = \mathbf{r}$;
- compute antidiffusive flux: $\mathbf{d} = (\mathbf{M}_l - \mathbf{M}_c)\mathbf{u}^h$;
- limit antidiffusive flux via FEM–FCT: $\mathbf{d}' = c_l \mathbf{d}$;
- add antidiffusive flux: $\mathbf{M}_l \mathbf{u} = \mathbf{M}_l \mathbf{u}^l + \mathbf{d}'$.

The bounds of the unknowns required to limit the antidiffusive fluxes are obtained by comparing the solution of mesh 2 to the solution of mesh 1 at all quadrature points contributing to the given point.

Conservative interpolation is of considerable importance for shock–structure interaction problems. As the surface grids of the fluid domain typically exhibit element sizes that are significantly smaller than the corresponding solid domain grids, straightforward interpolation can lead to numerical problems. The shock–plate interaction case shown in Figures 13.16–13.18 illustrates some of these. The discretizations employed for the structure and the fluid may be discerned in Figures 13.16(a) and (b). Note that the element size for the structural domain is considerably larger than that of the fluid domain.

The initial conditions for the flowfield are sketched in Figure 13.17(a). The surface pressure at a given time is shown in Figure 13.17(b). The corresponding pressure as seen by the structure using interpolation and the monotonicity preserving projection are shown in Figures 13.17(c) and (d). The total force exerted by the fluid and seen by the structure are compared in Figure 13.18. Note that the difference in forces can be significant if interpolation is used (in excess of 50%!), while the error for the (conservative) projection procedure vanishes.

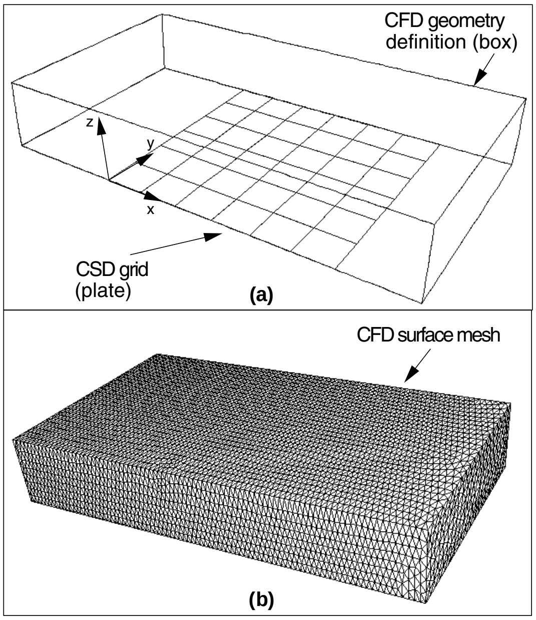


Figure 13.16. Shock on plate: (a) problem definition and (b) surface grids

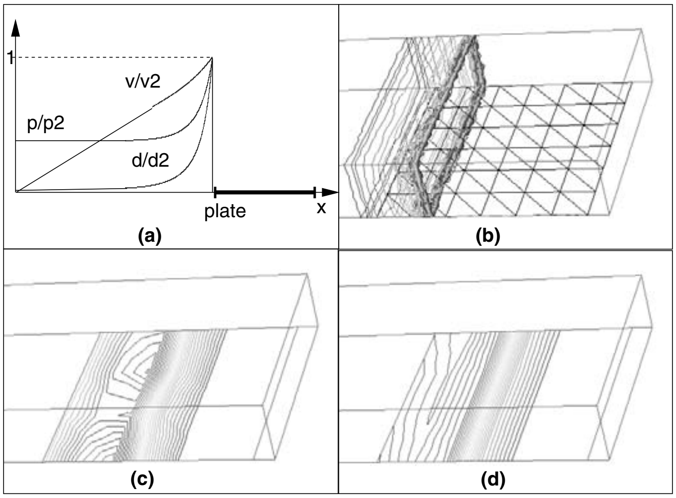


Figure 13.17. Shock on plate, results obtained: (a) detonation profile; (b) fluid pressure; (c) interpolation; (d) pFCT

13.7. Surface-grid-to-surface-grid interpolation

For many multi-disciplinary applications that employ so-called loose coupling techniques, a transfer of unknowns at the surfaces of the respective computational domains is required. The problem considered here is the fast interpolation of data between two surface triangulations.

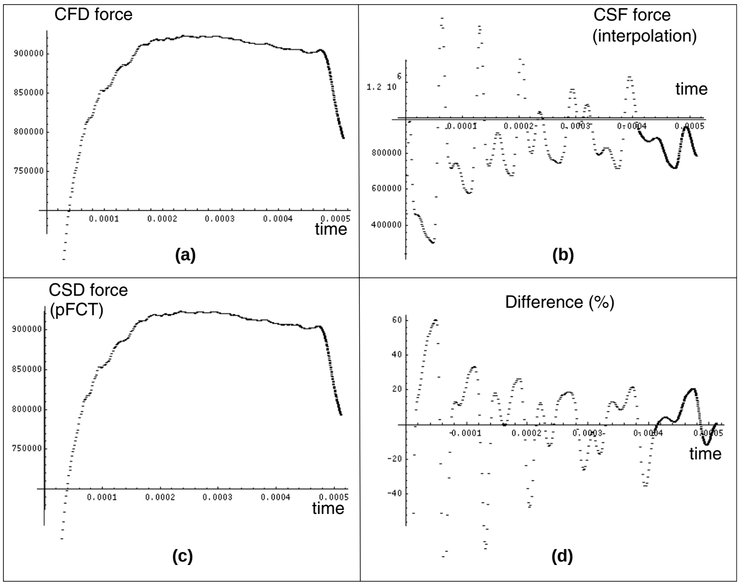


Figure 13.18. Shock on plate, comparison of forces

Other types of surface elements can be handled as described above. In what follows, we will refer to the triangular surface elements as faces. The basic idea is to treat the topology as 2-D, while the interpolation problem is given in 3-D space. This implies that further criteria, like relative distances normal to the surface, will have to be employed in order to make the problem unique. The basic procedure is to compute the shape functions of the surface triangles as before.

Using the notation of Figure 13.19, we can write

$$\mathbf{x}_p = \mathbf{x}_0 + \sum_i \alpha^i \mathbf{g}_i, \tag{13.20}$$

where

$$\mathbf{g}_{1,2} = \mathbf{x}_{1,2} - \mathbf{x}_0, \quad \mathbf{g}_3 = \frac{\mathbf{g}_1 \times \mathbf{g}_2}{|\mathbf{g}_1 \times \mathbf{g}_2|}, \tag{13.21a,b}$$

$$\alpha^{1,2} = N^{1,2}, \quad \alpha^0 = 1 - \alpha^1 - \alpha^2. \tag{13.21c,d}$$

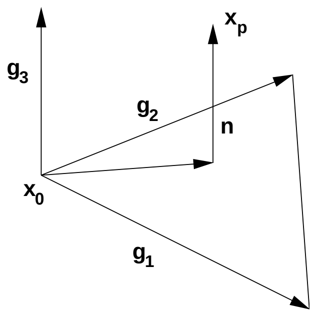
Point $\mathbf{x}_p$ may be considered as being on the surface face if

$$\min(N^i, 1 - N^i) \geq 0, \quad \forall i = 0, 1, 2, \tag{13.22a}$$

and

$$d_n = |\alpha^3 \mathbf{g}_3| < c_t \cdot |\mathbf{g}_1 \times \mathbf{g}_2|^{0.5}. \tag{13.22b}$$

Here c_t denotes a tolerance for the relative distance normal to the surface face. Even though this may be a problem-dependent variable, values of $c_t = O(0.05)$ work well. As before, an



.

Figure 13.19. Surface-to-surface interpolation

advancing-front neighbour-to-neighbour algorithm may be employed. The interpolation of surface grid information can be complicated by the fact that (13.9a) may never be satisfied (e.g. the convex ridge shown in Figure 13.20(a)), or may be satisfied by more than one surface face (e.g. the concave ridge shown in Figure 13.20(b)).

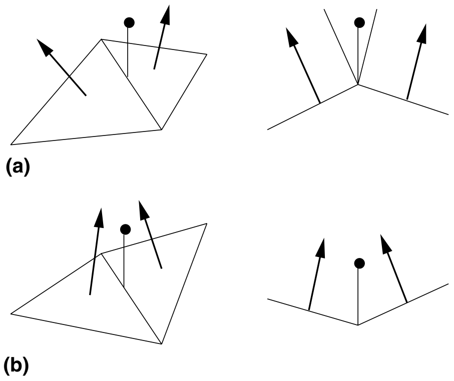


Figure 13.20. Surface-to-surface interpolation: (a) concave ridge (no host face); (b) convex ridge (multiple host faces)

In the first instance the criterion given by (13.9a) may be relaxed somewhat to

$$\min(N^i, 1 - N^i) > tol, \quad \forall i = 1, 2, 4, \tag{13.23}$$

where *tol* is a small number. For the second case, the surface face with the smallest normal distance d_n is selected. In both of these instances the interpolation error is unaffected by the final host surface face, as the interpolation weights are such that only the points belonging to the ridge are used for interpolation. For complex corners or some multi-body configurations an exhaustive search may be triggered. One is then faced with the decision as to which is the best host face. A procedure that was found effective is the following.

Consider all faces satisfying

$$\min(\alpha^i, 1 - \alpha^i) \geq \alpha_{es} < 0, \quad \forall i = 1, 2, 4 \tag{13.24a}$$

and

$$d_n = |\alpha^3 \mathbf{g}_3| \leq \delta_{es}. \tag{13.24b}$$

This implies a relaxed closeness criterion with $\alpha_{es} = -1$, $c_{es} = 0.5$. Then, keep the face closest to the point being interpolated:

- if (13.22a) is satisfied then the distance to the face is given by $\delta = d_n$;

- if (13.22a) is not satisfied, the closest distance to the face edges is taken:

$$\delta = \min_{ij} |\mathbf{x}_p - (1 - \beta_{ij})\mathbf{x}_i - \beta_{ij}\mathbf{x}_j|, \tag{13.25}$$

$$\beta_{ij} = \frac{(\mathbf{x}_p - \mathbf{x}_i) \cdot (\mathbf{x}_j - \mathbf{x}_i)}{(\mathbf{x}_j - \mathbf{x}_i) \cdot (\mathbf{x}_j - \mathbf{x}_i)}. \tag{13.26}$$

For the cases shown in Figure 13.21 we have the following.

Case 1: `IF1` satisfies (13.22a), but `IF2` is better.

Case 2: `IF1`, `IF2` have the same distance, implying that `IF2` is preferred as it satisfies (13.25a).

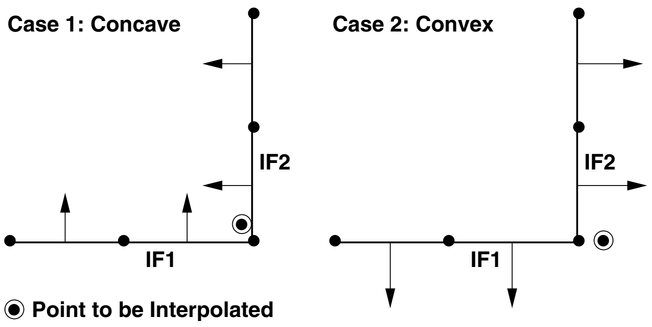


Figure 13.21. Interpolation to corners/ridges

In general, it is advisable to perform a local exhaustive search using all the faces surrounding the points of the host face found (see Figure 13.22) in order to determine the best host face.

Another common problem for fluid–structure interaction cases is the interpolation for shells or other 'thin surfaces'. For such cases, the closest face may be on the 'wrong side' (see Figure 13.23), leading to erroneous values in forces and displacements.

The most common solution to this dilemma is to define, for the smooth portions of the surface, a *point normal* $\mathbf{n}_p$. When interpolating, only faces that are aligned with this point normal are considered:

$$\mathbf{n}_f \cdot \mathbf{n}_p > c_s, \quad c_s = 0.5. \tag{13.27}$$

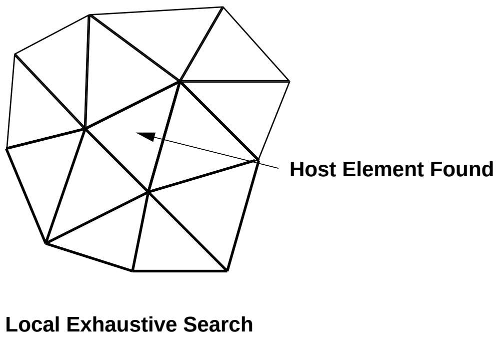


Figure 13.22. Neighbour elements for local exhaustive search

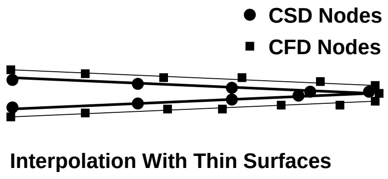


Figure 13.23. Surface–surface interpolation

13.8. Particle–grid interpolation

A large number of applications require the simultaneous representation of some part of the data in the form of grids and particles. Examples where this type of separation is commonly employed are as follows.

(a) *Particle-in-cell codes*. These types of codes, which are commonly used for plasma and particle beam simulations, sample the physical fields at Eulerian mesh points and represent certain aspects of transport by the motion of Lagrangian particles. The particles carry electric charge and mass, and interact through the Lorentz force equation with the electric and magnetic fields sampled from the grid. These fields in turn are updated in an Eulerian fashion using sources accumulated from the particles. During each timestep of the simulation, the particles change position. However, before they can contribute information to the grid or sample field information from it, their new host element or cell must be identified. Therefore, they must be traced through the grid. Given the large number of particles ($>10^6$) in typical PIC simulations, the need for fast interpolation algorithms becomes obvious.

(b) *Multiphase flow simulations with Lagrangian solid phase*. In many multiphysics flow simulations, it is advantageous to represent one part of the flow by a continuum (e.g. the mean gas flow), and the other by a set of particles (e.g. dust for shock–object simulations, burning particles in solid rocket exhaust plumes, etc.). The particles are accelerated by the gas through friction forces, and an exchange of mass, momentum and energy between the gas and the particles takes place. As the particles move through the grid, they must be traced, i.e. their host element must be identified. As before, the number of particles can easily exceed the number of elements in the mesh by an order of magnitude, making fast interpolation an enabling technology.

A common feature of all particle–grid applications is that the particles do not move far between timesteps. This makes physical sense: if a particle jumped ten gridpoints during one timestep, it would have no chance to exchange information with the points along the way, leading to serious errors. Therefore, the assumption that the new host elements of the particles are in the vicinity of the current ones is a valid one. For this reason, the most efficient way to search for the new host elements is via the vectorized neighbour-to-neighbour algorithm. The idea is to search for the host elements of as many particles as possible. The obstacle to this approach is that not every particle will satisfy criterion (13.5) in the same number of attempts or passes over the particles. As before, the solution is to reorder the particles to be interpolated after each pass so that all particles that have not yet found their host element are at the top of the list (Löhner (1990)).

A simple numerical example illustrates the speedup achievable by vectorizing the particle tracer as described above. The problem statement is as follows. Discretize the square region $[0, 1] \times [0, 1]$ using triangles. Then place a large number of particles in the mesh. Move all particles the same fixed distance, but in random directions. If a particle lies outside the $[0, 1] \times [0, 1]$ square, move it back inside. For the discretization, the mesh shown in Figure 13.24 was used.

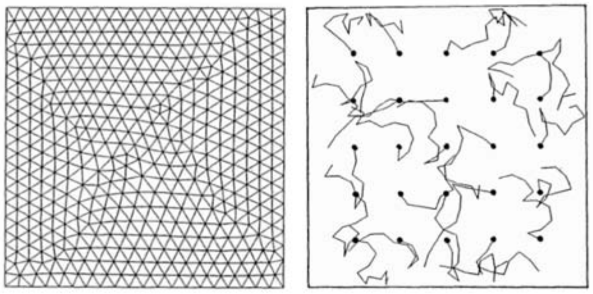


Figure 13.24. Particle tracing: mesh used and trajectories

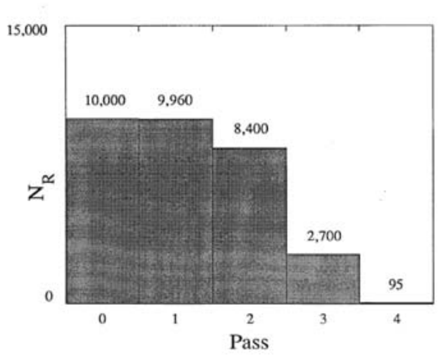


Figure 13.25. Histogram of tries required to find the host element

Table 13.2. Particle-mesh interpolation timings

	`ON=F,OFF=V`	`ON=F`
Particle-tracing time (s)	257.0	17.6
`MFLOP`-rate recorded	1.11	20.82
Total running time (s)	290.0	21.45
Timings (μs/particle/step)	51.4	3.52

At the beginning, the particles were placed uniformly across the mesh, following a square lattice of `nppdi`$\times$`nppdi` particles. Figure 13.24 shows a typical sequence of 20 steps for `npart=5`$\times$`5` particles. In an effort to obtain an accurate timing the following set of parameters was used:

- `npart=10,000`: this large set of particles minimizes the overhead associated with subroutine calls;
- `ntime=500`: this large number of timesteps minimizes the start-up overheads.

The particles were moved a distance of `dista=0.05` every timestep. On the mesh used, at most six elements had to be tested for each particle to locate the new host element. However, in most cases only five elements required testing.

The average number of remaining particles after each search pass, i.e. those that have not yet found their host elements, is shown in the histogram given by Figure 13.25. To obtain timings, two runs were performed on a Cray-XMP with and without automatic vectorization. The Cray-XMP allowed the required gather/scatter operations to be run in hardware. The timings obtained are listed in Table 13.2 below.

The results demonstrate a 14-fold speedup when using the automatic vectorization. Factors like these enable new dimensions of reality in simulations, making it possible to study new phenomena previously thought untractable.

14 ADAPTIVE MESH REFINEMENT

The development of self-adaptive mesh refinement techniques in computational fluid dynamics (CFD), computational structural dynamics (CSD), computational electromagnetics (CEM) and other fields of computational mechanics is motivated by a number of factors.

(a) With mesh adaptation, the numerical solution to a specific problem, given the basic accuracy of the solver and the desired accuracy, should be achieved with the fewest degrees of freedom. This, in most cases, translates into the least amount of work for a given accuracy. Work in this context should be understood as meaning not only CPU, but also memory and man-hour requirements.

(b) For some classes of problems, the savings in CPU and memory requirements exceed a factor of 100 (see Baum and Löhner (1989)). This is equivalent to two generations of supercomputing. For these problems, adaptive mesh refinement has acted as an enabling technology, allowing the simulation of previously intractable problems.

(c) Mesh adaptation avoids time delays incurred by trial and error in choosing a grid that is suitable for the problem at hand. Although this may seem unimportant for repetitive steady-state calculations, it is of the utmost importance for transient problems with travelling discontinuities (shocks, plastic fronts, etc.). In this way, adaptation adds a new dimension of user-friendliness to computational mechanics.

Given these very strong motivating reasons, the last two decades have seen a tremendous surge of activity in this area. It is interesting to note that mesh adaptation in CFD, CSD and CEM appeared in the early 1980s. In fact, the first conferences dedicated solely to this subject took place around 1984 (Babuska *et al.* (1983, 1986)). As with most of this book, the present chapter focuses primarily on unstructured, i.e. unordered, grids, such as those commonly encountered in finite element applications. However, parallel developments in the area of structured grids (finite difference and finite volume techniques) are mentioned where appropriate.

Any adaptive refinement scheme is composed of three main ingredients:

- an optimal-mesh criterion;
- an error indicator; and
- an algorithm or strategy to refine and coarsen the mesh.

They give answers to the questions:

- How should the optimal mesh be defined?

Applied Computational Fluid Dynamics Techniques: An Introduction Based on Finite Element Methods, Second Edition.
Rainald Löhner © 2008 John Wiley & Sons, Ltd. ISBN: 978-0-470-51907-3

- Where is refinement/coarsening required? and
- How should the refinement/coarsening be accomplished?

The topic of adaptation being now two decades old, it is not surprising that a variety of answers have been proposed by several authors for each of these questions. In the following, the most successful ones are discussed in more depth.

14.1. Optimal-mesh criteria

Before designing an adaptive mesh procedure, the analyst should have a clear idea of what is to be achieved. Reduction of manual and computational work is the obvious answer, but in order to be more definite one needs a quantitative assessment of the optimality of the adaptive mesh procedure. This leads to the immediate question: What should the optimal mesh be like? The answer to this crucial question is seldomly clear, as engineers do not always know *a priori* what constitutes a sufficiently accurate answer to the problem at hand. For example, if all that is required is the lift of an airfoil (an integrated, global quantity), there is in most cases no need for an extremely accurate solution away from the surface. A number of researchers have followed guidelines with a more rigorous mathematical foundation, which are outlined below.

(a) *Equidistribution of error.* The aim is to attain a grid in which the error is uniformly distributed in space. One can show that such a mesh has the smallest number of degrees of freedom (i.e. the smallest number of elements). Simply stated,

$$\epsilon^h \to \min, \quad \forall \mathbf{x} \in \Omega. \tag{14.1}$$

Conceptually, one can derive this criterion from the observation that for an initial mesh the error will have an irregular distribution as shown in Figure 14.1(a). If the number of degrees of freedom is kept the same, the distribution of element size and shape is all that may be changed. Given that the error varies with element size, smaller elements are required in regions of large errors, whereas larger elements may be employed in regions of small errors. After repositioning of points, the error distribution in space will become more regular, as shown in Figure 14.1(b). One can also see that the general aim stated in (14.1) will be achieved when the error is constant in the domain. This mesh optimality criterion is used most often in conjunction with steady-state problems.

(b) *Local absolute error tolerances.* In many applications, the required error tolerances may not be the same at all locations. Moreover, instead of using the general minimization stated in (14.1), one may desire to enforce absolute local bounds in certain regions of the domain:

$$\epsilon^h < c_1, \quad \forall \mathbf{x} \in \Omega_{\text{sub}}. \tag{14.2}$$

Mesh refinement or coarsening will then take place if the local error indicator exceeds or falls below given refinement or coarsening tolerances:

$$\epsilon^h > c_r \Rightarrow \text{refine}, \quad \epsilon^h < c_c \Rightarrow \text{coarsen}.$$

The calculation of skin-friction coefficients of airfoils is a possible application for such a mesh optimality criterion. The mesh optimality criterion based on local absolute error tolerance is most often used for transient problems.

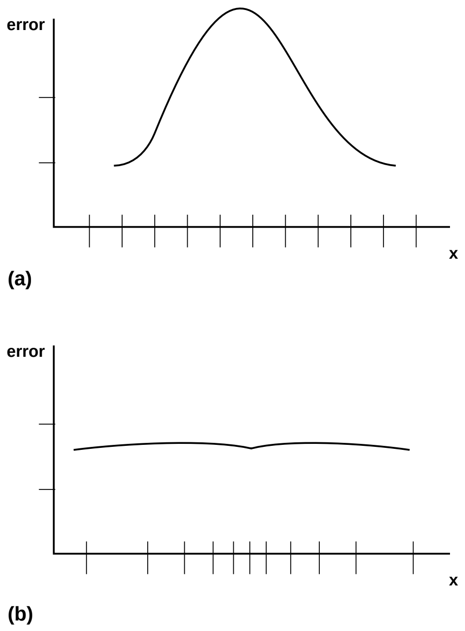


Figure 14.1. Equidistribution of error: (a) before adaptation; (b) after adaptation

14.2. Error indicators/estimators

Consider the task of trying to determine whether the solution obtained on the present mesh with the flow solver used (or any other field solver) is accurate. Intuitively, a number of criteria immediately come to mind: variations of key variables within elements, entropy levels, higher-order derivatives of the solutions, etc. All of them make the fundamental assumption that the solution on the present mesh is already in some form close to the exact solution. This assumption is reasonable for parabolic and elliptic problems, where, due to global minimization principles, local deficiencies in the mesh only have a local effect. For hyperbolic problems, the assumption $u^h \approx u$ may be completely erroneous. Consider an airfoil at high angle of attack. A coarse initial mesh may completely miss local separation bubbles at the leading edge that lead to massive separation in the back portion of the airfoil. Although the local error in the separation region may be very small (after all, the solution there is relatively smooth for a RANS simulation), the global error in the field would be very large due to different physical behaviour. *Any* local error indicator presently in use could miss these features for bifurcation point regions, performing adaptation at the wrong places. On the other hand, the assumption $u^h \approx u$ is a very reasonable one for most initial grids and stable physics. As a matter of fact, it is not so difficult to attain, given that the nature of most engineering applications is such that:

(a) developments are evolutionary, rather than revolutionary, i.e. a similar problem has been solved before (the airfoil being the extreme example); and

(b) the controllability requirement of reliable products implies operation in a stable regime of the physics.

14.2.1. ERROR INDICATORS COMMONLY USED

The most common error indicators presently used in production codes may be grouped into the following categories.

14.2.1.1. Jumps in indicator variables

This simplest error indicator is obtained by evaluating the jump (i.e. the absolute difference) of some indicator variable like the Mach number, density or entropy within an element or along an edge. This error indicator implicitly makes the assumption

$$\epsilon_{el}^{h} = c_1 h |\nabla u|, \tag{14.3}$$

i.e. first-order accuracy for the underlying scheme. Error indicators of this form have been used in industrial applications (Palmerio and Dervieux (1986), Dannenhoffer and Baron (1986), Kallinderis and Baron (1987), Mavriplis (1990b, 1991a), Aftosmis and Kroll (1991), DeZeeuw and Powell (1993)), even if the underlying numerical discretization was higher than first order.

14.2.1.2. Interpolation theory

Making the assumption that the solution is smooth, one may approximate the error in the elements by a derivative one order higher than the element shape function. For one dimension this would result in a local error indicator (i.e. at the element level) of the form

$$\epsilon_{el}^{h} = c_1 h^p \left| \frac{\partial^p u}{\partial x^p} \right|, \tag{14.4}$$

where $p-1$ is the polynomial degree of the shape functions and the pth derivative is obtained by some recovery procedure. The total error in the computational domain is then given by

$$\epsilon_{\Omega}^{h} = c_2 \int h^p \left| \frac{\partial^p u}{\partial x^p} \right| d\Omega. \tag{14.5}$$

This error indicator gives superior results for smooth regions. On the other hand, at discontinuities the local value of ϵ_{el}^{h} will stay the same no matter how fine the mesh is made. As a simple example, consider a shock across a fixed number of elements in one dimension. Assuming a grid of constant size h, the resulting error indicator at points will be of the form $|u_{i-1} - 2u_i + u_{i+1}|$. Observe that the element size has disappeared, i.e. no matter how fine the mesh is made, the error indicator values close to the discontinuity will remain unchanged. Furthermore, note that the global error will indeed decrease, but only as $O(h)$.

14.2.1.3. Comparison of derivatives

Again making the assumption that the solution is smooth, one may compare significant derivatives using schemes of different order. As an example, consider the following three approximations to a second derivative:

$$u_{,xx}|_{4_1} = \frac{1}{h^2}(u_{i-1} - 2u_i + u_{i+1}) - \frac{1}{12}h^2 u_{,IV}, \tag{14.6a}$$

$$u_{,xx}|_{4_2} = \frac{1}{4h^2}(u_{i-2} - 2u_i + u_{i+2}) - \frac{1}{12}4h^2 u_{,IV}, \tag{14.6b}$$

$$u_{,xx}|_6 = \frac{1}{12h^2}(-u_{i-2} + 16u_{i-1} - 30u_i + 16u_{i+1} - u_{i+2}) + \frac{1}{90}h^4 u_{,VI}. \tag{14.6c}$$

The assumption of smoothness in u would allow a good estimate of the error in the second derivatives from the difference of these three expressions (Schönauer *et al.* (1981)). Moreover, comparing (14.6a) to (14.6b,c) can give an indication as to whether it is more efficient to h-refine the mesh (reduction of h), or to increase the order of accuracy for the stencil (p-refinement). For unstructured grids, one may recover these derivatives with so-called recovery procedures (see Chapter 5).

14.2.1.4. *Residuals of PDEs on adjacent grids*

Assume we have a node-centred scheme to discretize the PDEs at hand. At steady state, the residuals at the nodes will vanish. On the other hand, if the residuals are evaluated at the element level, non-vanishing residuals are observed in the regions that require further refinement. This error indicator has been used extensively by Palmerio *et al.* (1985), and has a close link to so-called output-based indicators (see below). Another possibility is to check locally the effect of higher-order shape functions introduced at the element level or at element boundaries (Baehman *et al.* (1992)). These so-called p-refinement indicators have been used extensively for computational structural mechanics applications (Zienkiewicz *et al.* (1983), Babuska *et al.* (1983), Dunavant and Szabo (1983)).

14.2.1.5. *Energy norms*

Assume incompressible creeping flow. For this viscous-dominated flow the solution consists of a velocity field **v** with zero divergence that minimizes the dissipation energy functional

$$J(\mathbf{v}) = \int \mu(v^i_{,j} + v^j_{,i}) : (v^i_{,j} + v^j_{,i})\, d\Omega = \int \sigma : \epsilon \, d\Omega. \tag{14.7}$$

Here **v** denotes the velocity field, σ the viscous stress tensor and ϵ the strain rate tensor. The dissipation energy of the error $\mathbf{e} = \mathbf{v} - \mathbf{v}_h$ satisfies the relation

$$J(\mathbf{e}) = \int (\sigma_h - \sigma) : (\epsilon_h - \epsilon)\, d\Omega. \tag{14.8}$$

The unknown exact stresses and strain rates σ, ϵ can be recovered at nodes through a least-squares projection. The final estimator, using recovered stresses σ_r and strain rates ϵ_r, then becomes

$$J(\mathbf{e}) = \int (\sigma_h - \sigma_r) : (\epsilon_h - \epsilon_r)\, d\Omega. \tag{14.9}$$

This error estimator, although derived for creeping flows, has also been shown to be useful for flows with moderate Reynolds number (Hétu and Pelletier (1992)). One can also see that this error indicator is closely related to the Zienkiewicz–Zhu error indicator (Zhu and Zienkiewicz (1987), Zienkiewicz *et al.* (1988), Ainsworth *et al.* (1989), Wu *et al.* (1990)).

14.2.1.6. Energy of spatial modes

For higher-order methods, such as spectral element methods, a very elegant way to measure errors and convergence is to separate the energy contents associated with the different shape functions. The decrease of energy contained in the higher-order shape functions gives a reliable measure of convergence (Mavriplis 1990a, 1992). At the same time, this way of measuring errors provides an immediate strategy as to when to perform h-refinement (slow decrease of energy content with increasing shape-function polynomial) or p-refinement (rapid decrease of energy content with increasing shape-function polynomial).

14.2.1.7. Output-based error estimators

For many applications, the user is interested in a few global quantities. Typical examples are lift, drag and moments for aerodynamic problems. The idea is then to refine the grid only in those regions that affect these so-called outputs. This type of error estimator has received increased attention in recent years (Pierce and Giles (2000), Becker and Rannacher (2001), Rannacher (2001), Giles and Süli (2002), Süli and Houston (2002), Venditti and Darmofal (2002, 2003)). Assume a given numerical solution $\mathbf{u}^h$ with a numerical solution error $\delta\mathbf{u}$ so that the exact solution is given by

$$\mathbf{u} = \mathbf{u}^h + \delta\mathbf{u}. \tag{14.10}$$

Furthermore, consider a desired output function $I(\mathbf{u})$ (e.g. the drag of a wing) and physical constraints (e.g. the Navier–Stokes equations)

$$R(\mathbf{u}) = 0. \tag{14.11}$$

Performing a Taylor series expansion we have, for the desired output function and the constraints,

$$I(\mathbf{u}) = I(\mathbf{u}^h + \delta\mathbf{u}) = I(\mathbf{u}^h) + I_{,\mathbf{u}}|_{\mathbf{u}^h}\delta\mathbf{u} + \text{hot}, \tag{14.12}$$

$$R(\mathbf{u}) = R(\mathbf{u}^h + \delta\mathbf{u}) = R(\mathbf{u}^h) + R_{,\mathbf{u}}|_{\mathbf{u}^h}\delta\mathbf{u} + \text{hot} \tag{14.13}$$

where 'hot' stands for 'higher-order terms'. However, the current solution already satisfies $R(\mathbf{u}^h) = 0$, which implies

$$R(\mathbf{u}) \approx R_{,\mathbf{u}}|_{\mathbf{u}^h}\delta\mathbf{u}, \tag{14.14}$$

$$\delta\mathbf{u} \approx [R_{,\mathbf{u}}|_{\mathbf{u}^h}]^{-1}R(\mathbf{u}), \tag{14.15}$$

$$\delta I = I_{,\mathbf{u}}|_{\mathbf{u}^h}\delta\mathbf{u} \approx I_{,\mathbf{u}}|_{\mathbf{u}^h}[R_{,\mathbf{u}}|_{\mathbf{u}^h}]^{-1}R(\mathbf{u}) = \boldsymbol{\Psi}^T R(\mathbf{u}) \tag{14.16}$$

where $\boldsymbol{\Psi}$ denotes the *adjoint variables*, i.e. the solution of the system

$$[R_{,\mathbf{u}}|_{\mathbf{u}^h}]^T\boldsymbol{\Psi} = [I_{,\mathbf{u}}|_{\mathbf{u}^h}]^T. \tag{14.17}$$

Given that numerical errors are also present for $\mathbf{\Psi}^h$, one can assume that, as before for $\mathbf{u}$,

$$\mathbf{\Psi} = \mathbf{\Psi}^h + \delta\mathbf{\Psi}, \tag{14.18}$$

implying

$$\delta I = [\mathbf{\Psi}^h]^T R(\mathbf{u}) + [\delta\mathbf{\Psi}]^T R(\mathbf{u}). \tag{14.19}$$

The last term is of higher order, resulting in

$$I(\mathbf{u}) \approx I(\mathbf{u}^h) + [\mathbf{\Psi}^h]^T R(\mathbf{u}). \tag{14.20}$$

This last equation may be interpreted in two ways: first as a way to improve the computed values of $I(\mathbf{u})$; and secondly as a way to estimate where to refine the mesh. Note that the residual $R(\mathbf{u})$ (which may be used as an error indicator by itself) is multiplied by the adjoint $\mathbf{\Psi}$. The adjoint thus ‘weighs’ the influence or importance of a given residual in space with respect to the desired output function $I(\mathbf{u})$.

14.2.1.8. Other error indicators/estimators

In a field that is developing so rapidly, it is not surprising that a variety of other error indicators and estimators have been proposed. The more theoretically inclined reader may wish to consult Strouboulis and Oden (1990), Strouboulis and Haque (1992a,b) and Johnsson and Hansbo (1992).

14.2.2. PROBLEMS WITH MULTIPLE SCALES

All of these error indicators have been used in practice to guide mesh adaptation procedures. They all work well for their respective area of application. However, they cannot be considered as generally applicable for problems with multiple intensity and/or length scales. None of them are dimensionless, implying that strong features (e.g. strong shocks) produce large error indicator values, whereas weak features (such as secondary shocks, contact discontinuities, shear layers) produce small ones. Thus, in the end, only the strong features of the flow would be refined, losing the weak ones. A number of ways have been proposed to circumvent this shortcoming.

14.2.2.1. Stopping criteria

The most common way to alleviate the problems encountered with multiple intensity/length scales is to introduce a stopping criterion for refinement. This can be a minimum element size or, for the case of h-refinement, a maximum level of subdivision. After the discretization is fine enough to activate the stopping criterion, the remainder of the added degrees of freedom are introduced in regions with weaker features.

14.2.2.2. Two-pass strategy

A second option, proposed by Aftosmis and Kroll (1991), Aftosmis (1992), is to separate the disparate intensity or length scales in two (or possibly several) passes over the mesh. In the first pass, the strong features of the flow are considered, and appropriate action is taken.

A second pass is performed for the elements not marked in the first pass, identifying weak features of the flow, and appropriate action is taken. This procedure has worked well for viscous flows with shocks, and can be combined with any of the error indicators described above.

14.2.2.3. Non-dimensional error indicators

An error indicator that allows even refinement across a variety of intensity and length scales was proposed by Löhner (1987). In general terms, for linear elements it is of the form

$$\text{error} = \frac{h^2|\text{second derivatives}|}{h|\text{first derivatives}| + c_n|\text{mean value}|}. \tag{14.21}$$

Dividing the second derivatives by the absolute value of the first derivatives makes the error indicator bounded and dimensionless, and avoids the 'eating-up' effect of strong features. The terms following c_n are added as a noise filter in order not to refine wiggles or ripples which may appear due to loss of monotonicity. The value for c_n thus depends on the algorithm chosen to solve the PDEs describing the physical process at hand. The multi-dimensional form of this error indicator is given by

$$E^I = \sqrt{\frac{\sum_{k,l}(\int_\Omega N^I_{,k} N^J_{,l}\, d\Omega \cdot U_J)^2}{\sum_{k,l}(\int_\Omega |N^I_{,k}|[|N^J_{,l} U_J| + c_n(|N^J_{,l}||U_J|)]\, d\Omega)^2}}, \tag{14.22}$$

where N^I denotes the shape function of node I. The fact that this error indicator is dimensionless allows the simultaneous use of several indicator variables. Because the error indicator is bounded ($0 \le E^I \le 1$), it can be used for whole classes of problems without having to be scaled to the problem at hand. This results in an important increase in user-friendliness, allowing non-expert users access to automatic self-adaptive procedures. This error indicator has been used successfully for many years on a variety of applications (Löhner (1987, 1988b, 1989a,b), Baum and Löhner (1989), Löhner and Baum (1990), Löhner and Baum (1990), Baum and Löhner (1991, 1992), Löhner and Baum (1992), Loth *et al.* (1992), Sivier *et al.* (1992), Baum and Löhner (1993), Baum *et al.* (1994, 1995), Löhner *et al.* (2004c), Baum *et al.* (2006)).

14.2.3. DETERMINATION OF ELEMENT SIZE AND SHAPE

After the error in the present solution has been measured, or at least the regions of the computational domain that require further refinement or coarsening have been identified, the next question to be answered is the magnitude of mesh change required. This question is of minor importance for h-refinement or p-refinement, where the grid is simply subdivided further by factors of two in space or by adding the next higher degree polynomial to the space of available shape functions. On the other hand, for adaptive mesh movement or adaptive remeshing, where the element size and shape vary smoothly, it is necessary to obtain a more precise estimation of the required element size and shape. How to achieve this will be shown for the non-dimensional error indicator given by (14.21). It is a simple matter to perform a similar analysis for all the other error indicators described. Defining the derivatives according to order as

$$D_i^0 = c_n(|U_{i+1}| + 2 \cdot |U_i| + |U_{i-1}|), \tag{14.23a}$$

$$D_i^1 = |U_{i+1} - U_i| + |U_i - U_{i-1}|, \tag{14.23b}$$

$$D_i^2 = |U_{i+1} - 2 \cdot U_i + U_{i-1}|, \tag{14.23c}$$

the error indicator on the present (old) grid E^{old} is given by

$$E_i^{\text{old}} = \frac{D_i^2}{D_i^1 + D_i^0}. \tag{14.24}$$

The reduction of the current element size h^{old} by a fraction ξ to $h^{\text{new}} = \xi h^{\text{old}}$ will yield a new error indicator of the form

$$E_i^{\text{new}} = \frac{D_i^2 \xi^2}{D_i^1 \xi + D_i^0}. \tag{14.25}$$

Given the desired error indicator value E^{new} for the improved mesh, the reduction factor ξ is given by

$$\xi = \frac{E^{\text{new}}}{2E^{\text{old}}\left[\frac{D_i^1 + \sqrt{(D_i^1)^2 + 4D_i^0 \frac{E^{\text{old}}}{E^{\text{new}}}[D_i^1 + D_i^0]}}{[D_i^1 + D_i^0]}\right]}. \tag{14.26}$$

Observe that for a smooth solution with $D^1 \ll D^0$ this results in $\xi = (E^{\text{new}}/E^{\text{old}})^{0.5}$, consistent with the second-order accuracy of linear elements. Close to discontinuities $D^1 \gg D^0$ holds, and one obtains $\xi = E^{\text{new}}/E^{\text{old}}$, consistent with the first-order error obtained in these regions.

This error indicator can be generalized to multi-dimensional situations by defining the following tensors:

$$(D^0)_{kl}^I = h^2 c_n \int_\Omega |N_{,k}^I||N_{,l}^J||U_J| \, d\Omega, \tag{14.27}$$

$$(D^1)_{kl}^I = h^2 \int_\Omega |N_{,k}^I||N_{,l}^J U_J| \, d\Omega, \quad (D^2)_{kl}^I = h^2 \left| \int_\Omega N_{,k}^I N_{,l}^J \, d\Omega U_J \right|, \tag{14.28}$$

which yield an error matrix $\mathbf{E}$ of the form

$$\mathbf{E} = \begin{Bmatrix} E_{xx} & E_{yx} & E_{zx} \\ E_{xy} & E_{yy} & E_{zy} \\ E_{xz} & E_{yz} & E_{zz} \end{Bmatrix} = \mathbf{X} \cdot \begin{Bmatrix} E_{11} & 0 & 0 \\ 0 & E_{22} & 0 \\ 0 & 0 & E_{33} \end{Bmatrix} \cdot \mathbf{X}^{-1}. \tag{14.29}$$

The principal eigenvalues of this matrix are then used to obtain reduction parameters $\xi_{j'j'}$ in the three associated eigenvector directions. Due to the symmetry of $\mathbf{E}$, this is an orthogonal system of eigenvectors that defines a local coordinate system.

The variation in element size required to meet a certain tolerance can be determined with any of the error indicators enumerated in section 14.1. However, the variation in stretching, i.e. the shape of the elements for the adapted 3-D mesh, requires an error indicator that is based on a tensor of second derivatives.

14.3. Refinement strategies

Besides the mesh optimality criterion and the error indicator/estimator, the third ingredient of any adaptive refinement method is the refinement strategy, i.e. *how to refine* a given mesh. Three different families of refinement strategies have been considered to date.

14.3.1. MESH MOVEMENT OR REPOSITIONING (R-METHODS)

The aim is to reposition the points in the field without changing the element topology (point connectivity), in order to obtain a better discretization for the problem at hand. The regions that require more elements tend to draw points and elements from regions where a coarser mesh can be tolerated. Three basic approaches have been used to date:

(a) the moving finite element method, where the position of points is viewed as a further unknown in a general functional to be minimized (Miller and Miller (1981));

(b) spring systems, whereby the mesh is viewed as a system of springs whose stiffness is proportional to the error indicator (Diaz *et al.* (1983), Gnoffo (1983), Nakahashi and Deiwert (1985), Palmerio *et al.* (1985), Palmeiro and Dervieux (1986)); and

(c) optimization methods, whereby the position of points is changed in order to minimize a functional (Brackbill and Saltzman (1982), Jacquotte and Cabello (1990)).

Because the mesh topology is not allowed to change, mesh movement schemes are relatively simple to code. They have the desirable property of aligning elements with features of lower dimensionality than the problem at hand. This stretching of elements can lead to considerable savings as compared to other methods. On the other hand, they are not flexible and general enough for production runs that may exhibit complex physics. They are presently used mainly in conjunction with finite difference codes (Gnoffo (1983), Carcaillet *et al.* (1983), Nakahasi and Deiwert (1985), Thompson (1985), Jacquotte and Cabello (1990)), where, by the very nature of the method, no topology changes are allowed. For unstructured grid codes, mesh movement has only been tried in academia or in conjunction with other methods (Palmerio and Dervieux (1986)) (see section 14.3.4 below).

14.3.2. MESH ENRICHMENT (H/P-METHODS)

In this case, degrees of freedom are added or taken from a mesh. One may either split elements into new ones (h-refinement, see Figure 14.2(a)), or add further degrees of freedom with hierarchical shape functions (Figure 14.2(b)). The same may be accomplished with the addition of higher-order shape functions (p-refinement), again either conventional polynomials (Babuska *et al.* (1986)), spectral functions (Mavriplis (1990a, 1992)) or hierarchical shape functions (Zienkiewicz *et al.* (1983), see Figure 14.2(c)).

For elliptic systems of PDEs, the combination of h- and p-refinement leads to exponential convergence rates (Babuska *et al.* (1986), Devloo *et al.* (1988)). P-refinement methods have so far not been used extensively in fluid mechanics. The author is not aware of any production or commercial CFD code that has a working built-in p-refinement capability. Possible reasons for this lack of success, which stands in contrast to CSD applications (Dunavant and Szabo (1983), Wang *et al.* (1984), Szabo (1986), Babuska *et al.* (1986)) are the following.

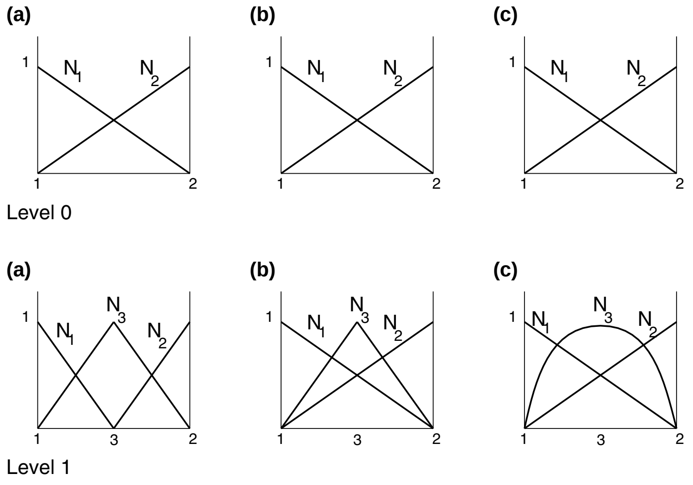


Figure 14.2. Mesh enrichment

(a) The limited accuracy that is achievable for CFD applications due to monotonicity enforcement close to discontinuities (Godunov (1959)), and the lack of accurate turbulence models (Anderson and Bonhaus (1993), Hystopolulos and Simpson (1993)). Indeed, a CFD solution that claims an accuracy better than 1% for a complex flow problem has to be considered with great scepticism. On the other hand, the major gains of high-order methods as compared to h-refinement in combination with low-order methods are in the range below 1% relative error.

(b) The desired accuracy of engineering applications, which seldom fall below 1% relative error. Given the uncertainties in boundary conditions, material parameters and source terms of typical engineering applications, 1% relative error can already be considered unnecessarily accurate.

(c) The much higher coding complexity of p-methods or h/p-methods as compared to straightforward h-methods. The only possible difficulty of h-methods is given by hanging nodes for quad- or brick-type elements (for triangles and tetrahedra even this problem disappears). Apart from this relatively small modification, the original one-element-type flow solver can remain unchanged. On the other hand, any p-method requires the development and maintenance of a complete library for all the different types of elements. Further complications arise from the adaptation logic when h/p-type refinements are considered.

This is not to say that we may not see much activity in this area: boundary layers, shear layers, flames and other features that are currently being computed using under-resolved Navier–Stokes runs are, when properly resolved, smooth features that should be ideally suited to high-order discretizations.

14.3.2.1. H-enrichment

By far the most successful mesh enrichment strategy has been h-enrichment. There are several reasons that can be given for this success.

- Conservation is maintained naturally with h-refinement.
- No interpolations other than the ones naturally given by the element shape functions are required. Therefore, no numerical diffusion is introduced by the adaptive refinement procedure. This is in contrast to adaptive remeshing, where the grids before and after a mesh change may not have the same points in common. The required interpolations of the unknowns will result in an increased amount of numerical diffusion (see Löhner (1988b, 1989b, 1990)).
- H-refinement is very well suited to vector and parallel processors. This is of particular importance for transient problems, where a mesh change is performed every 5 to 10 timesteps, and a large percentage of mesh points is affected in each mesh change (Löhner (1987), Löhner and Baum (1992), Fursenko (1993), Biswas and Strawn (1993)).

The three main variants used to date achieve mesh enrichment/coarsening (see Figure 14.3) by:

(a) classic subdivision of elements into four (2-D) or eight (3-D) after compatibility tests;

(b) recursive subdivision of the largest edge side with subsequent compatibility tests;

(c) agglomeration of cells with blocked subdivisions after compatibility tests.

The compatibility tests are necessary to ensure that an orderly transition occurs between fine and coarse mesh regions, and to avoid hanging nodes for triangular and tetrahedral elements.

Classic h-refinement

This simplest form of mesh refinement/ coarsening seeks to achieve higher mesh densities by subdividing elements into four (2-D) or eight (3-D). In order to have a smooth transition between refined and unrefined mesh regions, compatibility tests are required. For triangles and tetrahedra, badly formed elements due to refinement are avoided by restricting the possible refinement patterns. For tetrahedra, subdivision is only allowed into two (along a side), four (along a face) or eight. These cases are denoted as 1:2, 1:4 and 1:8, respectively. At the same time, a 1:2 or 1:4 tetrahedron can only be refined further to a 1:4 tetrahedron, or by first going back to a 1:8 tetrahedron with subsequent further refinement of the eight sub-elements. These are denoted as 2:4, 2:8+ and 4:8+ refinement cases. The refinement cases are summarized in Figure 14.4. This restrictive set of refinement rules avoids the appearance of deformed elements. At the same time, it considerably simplifies the refinement/coarsening logic. An interesting phenomenon that does not appear in two dimensions is the apparently free choice of the inner diagonal for the 1:8 refinement case. As shown in Figure 14.5, one can place the inner four elements around the inner diagonals 5–10, 6–8 or 7–9. Choosing the shortest inner diagonal produces the fewest distorted tetrahedra in the refined grid. When coarsening, again only a limited number of cases that are compatible with refinement

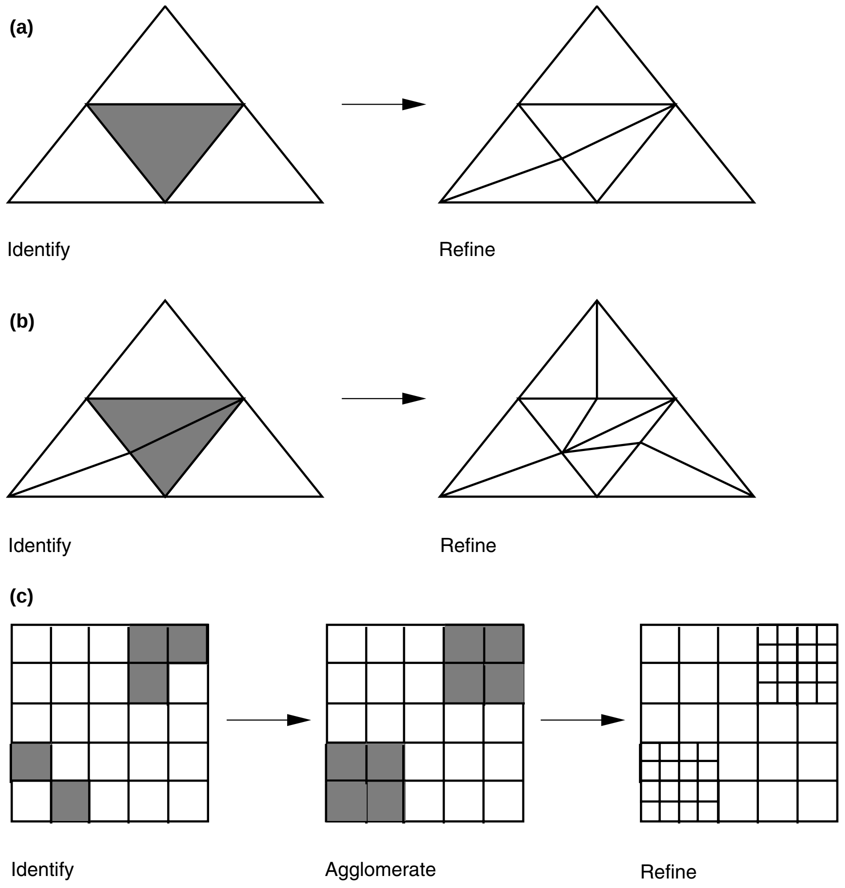


Figure 14.3. Possible h-refinement strategies

is allowed. The coarsening cases become 8:4, 8:2, 8:1, 4:2, 4:1 and 2:1 (Figure 14.6). Similar possible refinement patterns have been devised for triangles (Löhner (1987), Batina (1990b)), quads (Dannenhoffer and Baron (1986), Oden *et al.* (1986), Kallinderis and Baron (1987), Shaphiro and Murman (1988), Davis and Dannenhoffer (1989), Aftosmis and Kroll (1991)) bricks (Shaphiro and Murman (1988), Aftosmis (1992), Bieterman *et al.* (1992), Davis and Dannenhoffer (1993)) and mixed meshes (Mavriplis (1997)).

A considerable simplification in logic can be achieved by limiting to one the number of refinement/coarsening levels per mesh change (Löhner (1987), Löhner and Baum (1992)).

The main algorithmic steps then become:

- identify the elements to be refined/coarsened;
- make elements to be refined compatible by expanding the refinement region;
- make elements to be coarsened compatible by reducing the coarsening region;
- refine/coarsen the mesh;

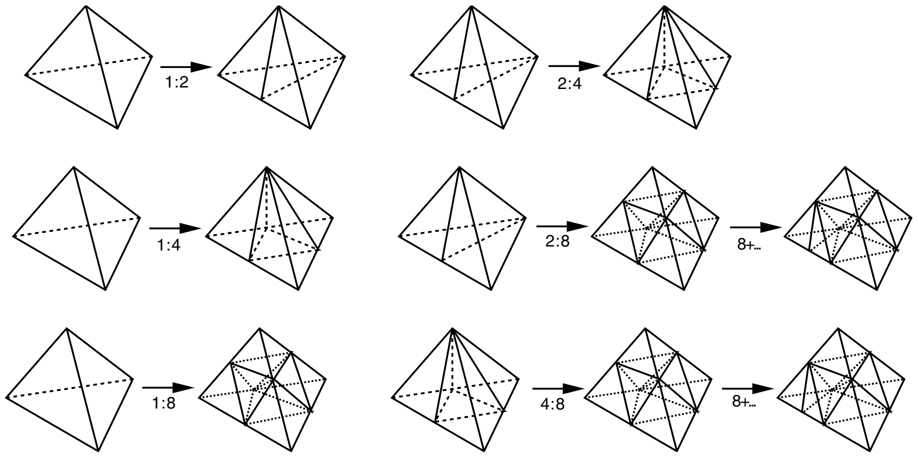


Figure 14.4. Refinement cases for tetrahedra

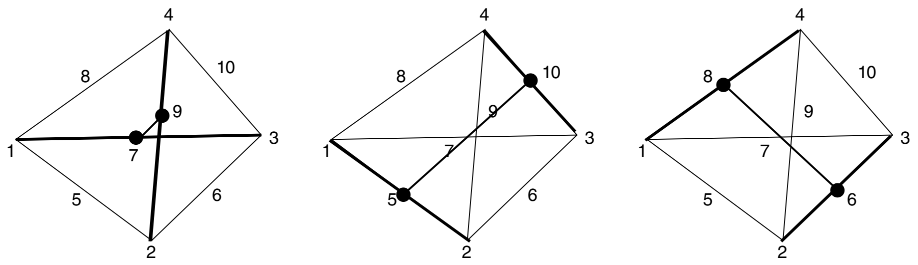


Figure 14.5. Choice of inner diagonal

- correct the location of new boundary points according to the surface definition data available;

- interpolate the unknowns and boundary conditions.

Most of the steps listed above are easily vectorizable and parallelizable, leading to an extremely fast adaptation procedure. This makes h-refinement one of the few adaptation procedures suitable for strongly unsteady flows (Baum and Löhner (1989, 1991, 1992), Sivier *et al.* (1992)). H-refinement has had the biggest impact in the field, leading to saving ratios in CPU and memory requirements in excess of 1:100 (Baum (1989)), the equivalent of two supercomputer generations. It has allowed the routine simulation of problems that had been intractable to date, opening new areas of applications for CFD (Baum (1989), Löhner and Baum (1990), Baum and Löhner (1991, 1992), Löhner and Baum (1992), Sivier *et al.* (1992), Baum and Löhner (1993), Baum *et al.* (1994, 1995), Löhner *et al.* (2004c), Baum *et al.* (2006)).

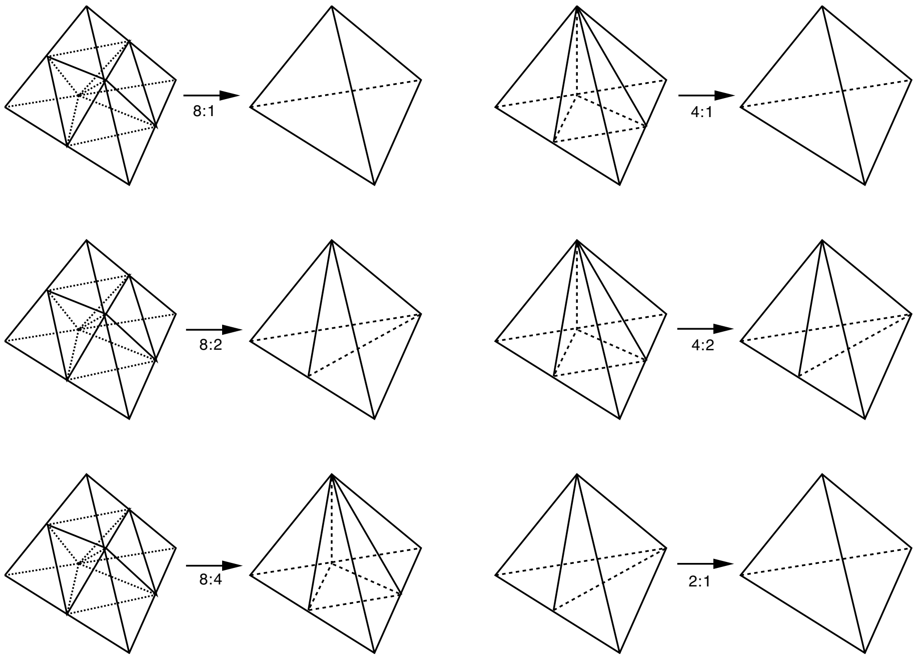


Figure 14.6. Coarsening cases for tetrahedra

Recursive subdivision of the largest edge

The separate coding and maintenance of allowable refinement and coarsening patterns observed for classic h-refinement in conjunction with triangles and tetrahedra can be avoided by using a recursive subdivision algorithm proposed by Rivara (1984, 1990, 1992). Whenever an element has been marked for refinement, the largest edge of the element is subdivided, producing the 1:2 refinement pattern in Figure 14.3(b). Hanging nodes are then reconnected to produce a mesh without hanging nodes. The elements are checked again for refinement, and the procedure is repeated until a fine enough mesh has been obtained. This algorithm may be summarized as follows.

Until the expected error is decreased sufficiently:

- identify the elements to be refined/coarsened;
- subdivide the largest edge of all elements marked for refinement;
- subdivide further elements with hanging nodes until no hanging nodes are present.

Agglomeration of cells with blocked subdivisions

The main aim of this type of refinement, which so far has been used only with quad- and brick-elements, is to achieve locally ordered blocks of data, so that the number of (expensive) indirect addressing operations required by the flow solver can be minimized. The regions to be refined or coarsened are agglomerated in such a way that blocks form. These blocks are then treated as separate grids within a multiblock code. A compromise has to be struck between too many small blocks (which minimize the total number of elements) and too few large blocks (which minimize the number of blocks and inter-block communication). The main algorithmic steps can be summarized as follows:

- identify the elements to be refined/coarsened;
- identify possible blocking patterns;
- make blocking patterns compatible;
- refine/coarsen the mesh;
- correct the location of new boundary points according to the surface definition data available;
- interpolate the unknowns and boundary conditions.

As one can see, the main difference between this algorithm and the classic h-refinement lies in the pattern recognition and compatibility checking phases that are now operating on blocks instead of individual elements. In particular, the pattern recognition phase has been the subject of considerable research (Berger and Oliger (1984)), producing impressive results (Berger and LeVeque (1989), Davis and Dannenhoffer (1993), Quirk (1994)).

14.3.3. ADAPTIVE REMESHING (M-METHODS)

This third family of refinement strategies only came into existence after the development of automatic grid generators. The grid generator is used in combination with an error indicator based on the present discretization and solution to remesh the computational domain either globally or locally, in order to produce a more suitable discretization. The main advantages offered by adaptive remeshing methods are the following.

(a) The possibility of stretching elements when adapting to features that are of lower dimensionality than the problem at hand (e.g. shock surfaces in 3-D), which leads to considerable savings as compared to h-refinement.

(b) The ability to accommodate in a straightforward manner problems with moving bodies, free surfaces or changing topologies. For this class of problems a fixed mesh structure will, in most cases, lead to badly distorted elements. This implies that at least a partial regeneration of the computational domain is required. An observation made from practical runs is that, as the bodies move through the flowfield, the positions of relevant flow features will change. Therefore, in most of the computational domain, a new mesh distribution will be required.

Any of the automatic grid generation techniques currently available – advancing front (Peraire *et al.* (1987), Löhner (1988a), Löhner and Parikh (1988), Peraire (1990), Löhner *et al.* (1992)), Delaunay triangulations (Baker (1989), Mavriplis (1990b, 1991a)) and modified quadtree/octree (Yerri and Shepard (1984), Shepard and Georges (1991)) – may be employed to regenerate the mesh. The steps required for one *adaptive remeshing* are as follows:

- obtain the error indicator matrix for the gridpoints of the present grid;
- given the error indicator matrix, get the element size, element stretching and stretching direction for the new grid;
- using the old grid as the background grid, remesh the computational domain;
- if further levels of global h-refinement are desired, refine the new grid globally;
- interpolate the solution from the old grid to the new one.

For adaptive remeshing using the advancing front method, see Peraire *et al.* (1987), Löhner (1988b,c, 1989b, 1990), Peraire *et al.* (1990), Tilch (1991), Probert *et al.* (1991), Hétu and Pelletier (1992), Loth *et al.* (1992), Huan and Wu (1992), Strouboulis and Haque (1992b), Peraire *et al.* (1992b), Baum and Löhner (1993); using Delaunay triangulations, see Mavriplis (1990, 1991a), Hassan *et al.* (1998); and using the modified quad/octree approach, see Baehman *et al.* (1992).

14.3.3.1. Local remeshing

For some classes of problems, e.g. subsonic store separation, the mesh may become distorted at a rate that is higher than the movement of physically relevant features. Another observation made from practical simulations is that badly distorted elements appear more frequently than expected from the element size prescribed. Given the relatively high cost of global remeshing, local remeshing in the vicinity of the distorted elements becomes an attractive option for these situations. The steps required are as follows:

- identify the badly distorted elements in the layers that move, writing them into a list `lerem(1:nerem)`;
- add to this list the elements surrounding these badly distorted elements;
- form holes in the present mesh by:
    - forming a new background mesh with the elements stored in the list `lerem`,
    - deleting the elements stored in `lerem` from the current mesh,
    - removing all unused points from the grid thus obtained;
- recompute the error indicators and new element distribution for the background grid;
- regrid the holes.

Typically, only a very small number of elements (<10) becomes so distorted that a remeshing is required. Thus, local remeshing is a very economical tool that allows reductions in CPU requirements by more than 60% for typical runs (Löhner (1990), Baum and Löhner (1993), Baum *et al.* (1994, 1995)).

14.3.4. COMBINATIONS

From the characteristics listed above, it is clear that all of the possible mesh refinement strategies exhibit weaknesses. Not surprisingly, some hybrid methods have been pursued that seek to compensate the weaknesses of the individual methods with their respective strengths. The most common are the following.

(a) *Mesh enrichment followed by mesh movement.* In this approach, which has been used mainly for steady-state applications (Palmerio *et al.* (1986)), the physical complexity is first accounted for by classic h-enrichment. The mesh is then further improved by moving the points. In this way, the flexibility of the h-enrichment technique is combined with the directionality feature of mesh movement.

(b) *Mesh regeneration and mesh enrichment.* In this approach, which has been used for strongly unsteady problems with moving bodies (Winsor *et al.* (1991), Baum *et al.* (1994)), the physical complexity is treated by classic h-enrichment. Moving bodies and/or changing topologies are treated by remeshing. By using h-enrichment as the main adaptive strategy, the number of remeshings is kept to a minimum. This in turn minimizes the loss of information and physics due to interpolation when remeshing. Moreover, local remeshing is used whenever possible. After a new coarse mesh has been obtained, it is subsequently h-enriched. At each stage of this process, the solution before remeshing is interpolated to the new h-enriched mesh. As most of the points in the mesh will coincide, the loss of information and physics is again kept to a minimum.

14.4. Tutorial: h-refinement with tetrahedra

Due to its widespread use in production codes, it seems appropriate to devote a section to a more thorough description of classic h-enrichment with tetrahedral elements. The reader may also get an indication of what it takes to get any of the techniques described above to work. As stated above, the number of refinement/coarsening levels per mesh change is limited to one. Moreover, refinement of a tetrahedron is only allowed into two (along a side), four (along a face) or eight new tetrahedra. These cases are denoted as 1:2, 1:4 and 1:8, respectively. At the same time, a 1:2 or 1:4 tetrahedron can only be refined further to a 1:4 tetrahedron, or by first going back to a 1:8 tetrahedron with subsequent further refinement of the eight sub-elements. These are the so-called 2:4, 2:8+ and 4:8+ refinement cases. The refinement cases are summarized in Figure 14.4. This restrictive set of refinement rules is necessary to avoid the appearance of ill-deformed elements. At the same time, it considerably simplifies the refinement/coarsening logic. For coarsening, again only a limited number of cases that are compatible with the refinement is allowed. Thus, the coarsening cases become 8:4, 8:2, 8:1, 4:2, 4:1 and 2:1. These coarsening cases are summarized in Figure 14.6.

When constructing the algorithm to refine or coarsen the grid one faces the usual choice of speed versus storage. The more information from the previous grid is stored, the faster the new grid may be constructed. One possible choice is to use a modified tree structure which requires 12 integer locations per element in order to identify the ‘parent’ and ‘son’ elements of any element, as well as the element type.

The first *seven* integers store the new elements (‘sons’) of an element that has been subdivided into eight (1:8). For the 1:4 and 1:2 cases, the sons are also stored in this allocated space, and the remaining integer locations are set to zero.

In the *eighth* integer, the element from which the present element originated (the ‘parent’ element) is stored.

The *ninth* integer denotes the position number in the parent element from which this element came.

The *tenth* integer denotes the element type. One can either have parents or sons of 1:8, 1:4 or 1:2 tetrahedra. These are marked by a positive value of the element type for the parents and a negative value for the sons. Thus, for example, the son of a 1:8 element would be marked as -8.

Finally, in the *11th* and *12th* integer locations the local and global refinement levels are stored.

These 12 integer locations per element are sufficient to construct further refinements or to reconstruct the original grid. It is clear that in these 12 integers a certain degree of redundancy is present. For example, the information stored in the tenth integer could be recovered from the data stored in locations 1:8 and 11:12. However, this would require a number of non-vectorizable loops with many `IF`-tests. Therefore, it was decided to store this value at the time of creation of new elements instead of recomputing it at a later time. Similarly, the 11th integer can be recovered from the information stored in locations 1:8 and 12. Again, storage was traded for CPU time.

14.4.1. ALGORITHMIC IMPLEMENTATION

Now that the basic refinement/coarsening strategy has been outlined, its algorithmic implementation can be described in more depth. One complete grid change requires the following five steps:

1. construction of the missing grid information needed for a mesh change (basically the sides of the mesh and the sides adjoining each element);
2. identification of the elements to be refined;
3. identification of the elements to be deleted;
4. refinement of the grid where needed;
5. coarsening of the grid where needed.

14.4.1.1. Construction of missing grid information

The missing information consists of the sides of the mesh and the sides belonging to each element. The sides are dynamically stored in a linked list, i.e. in two arrays, one containing the two points each side connects and the other one (a pointer array) containing the lowest side number reaching out of a point. The formation of these two arrays has been treated in Chapter 2. After these two side arrays have been formed, a further loop over the elements is performed, identifying which sides belong to each element.

14.4.1.2. Identification of elements to be refined

The aim of this sub-step is to determine on which *sides* further gridpoints need to be introduced, so that the resulting refinement patterns on an element level belong to the allowed cases listed above, thus producing a compatible, valid new mesh. Five main steps are necessary to achieve this goal:

(i) mark elements that require refinement;

(ii) add protective layers of elements to be refined;

(iii) avoid elements that become too small or that have been refined too often;

(iv) obtain preliminary list of sides where new points will be introduced;

(v) add further sides to this list until an admissible refinement pattern is achieved.

The first three of these steps are obvious. The last two are explained in more detail.

(iv) *Obtain preliminary list of sides for new points*. Given the side/element information obtained in section 14.4.1.1, one can determine a first set of sides on which new gridpoints need to be introduced. This set of sides is still preliminary, as only certain types of refinement are allowed.

(v) *Add further sides to this list until an admissible refinement pattern has been achieved.* The list of sides marked for the introduction of new points is still preliminary at this point. In most cases, it will not lead to a refinement pattern that only admits the cases described above (1:2, 1:4, 1:8, etc.). Therefore, further sides are marked for the introduction of new points until an admissible refinement pattern is reached. This is accomplished by looping several times over the elements, checking on an element level whether the set of sides marked can lead to an admissible new set of sub-elements. The algorithm used is based on the observation that the admissible cases are based on the introduction of new points along one side (1:2), three contiguous sides (1:4) or six contiguous sides (1:8). These admissible cases can be obtained from the following element-by-element algorithm (see Figure 14.7):

- set the node-array `lnode(1:4)=0`;
- loop over the sides of the element: if the side has been marked for the introduction of a new point, set `lnode(ip1)=1`, `lnode(ip2)=1`, where `ip1`, `ip2` are the end-nodes corresponding to this side;
- loop over the sides of the element: if `lnode(ip1)=1` and `lnode(ip2)=1`, mark the side marked for the introduction of a new point.

Calculations with several admissible layers of refinement and large grids reveal that sometimes up to 15 passes over the mesh are required to obtain an admissible set of sides. This relatively high number of passes can occur when the mesh exhibits regions where the refinement criterion is only just met by the elements. Then, the list of sides originally marked for refinement will be far from an admissible one. In each pass over the mesh, a further ‘layer’ of elements with admissible sides marked for refinement will be added. Moreover, as an element can be refined in six possible ways, in some cases it may take three passes to go from a 1:2 to a 1:8 case. Thus, the ‘front’ of elements with an admissible set of sides marked for refinement may advance slowly, resulting in many passes over the mesh. A considerable reduction in CPU is realized by presorting the elements as follows:

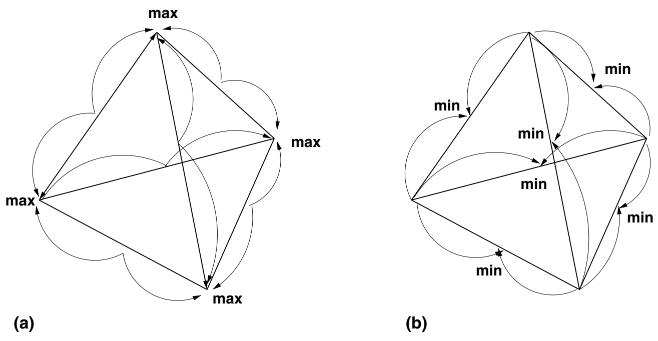


Figure 14.7. Fast compatibility checking algorithm: (a) sides to points; (b) points to sides

- add up all the sides marked for refinement in an element;
- if zero, one or six sides were marked do not consider further;
- if four or five sides were marked mark all sides of this element to be refined;
- if two or three sides were marked analyse in depth as described above.

This then yields the final set of sides on which new gridpoints are introduced.

14.4.1.3. Identification of elements to be deleted

The aim of this sub-step is to determine which *points* are to be deleted, so that the resulting coarsening patterns on an element level belong to the allowed cases listed above, thus producing a compatible, valid new mesh. Four main steps are necessary to achieve this goal:

(i) mark elements to be deleted;

(ii) filter out elements where father and all sons are to be deleted;

(iii) obtain preliminary list of points to be deleted;

(iv) delete points from this list until an admissible coarsening pattern is achieved.

The first two of these steps are obvious. The last two are explained in more detail.

(iii) *Obtain preliminary list of points to be deleted*. Given the list of parent elements to be coarsened, one can now determine a preliminary list of points to be deleted. Thus, all the points that would be deleted if all the elements contained in this list were coarsened are marked as ‘total deletion points’.

(iv) *Delete points from this list until an admissible coarsening pattern has been achieved*. The list of total deletion points obtained in the previous step is only preliminary, as unallowed coarsening cases may appear on an element level. Therefore loops are performed over the elements, deleting all those total deletion points which would result in unallowed coarsening cases for the elements adjoining them. This process is stopped when no incompatible total deletion points are left. As before, this process may be made considerably faster by grouping together and treating differently the parent elements with 0, 1, 2, 3, 4, 5 or 6 total deletion points.

14.4.1.4. Refinement of the grid where needed

New points and elements are introduced in two independent steps, which in principle could be performed in parallel.

(i) *Points*. To add further points, the sides marked for refinement in 2 are grouped together. For each of these sides a new gridpoint will be introduced. The coordinates and unknowns are interpolated using the side/point information obtained before. These new coordinates and unknowns are added to their respective arrays. In the same way new boundary conditions are introduced where required, and the location of new boundary points is adjusted using the CAD–CAM data defining the computational domain.

(ii) *Elements*. In order to add further elements, the sides marked for refinement are labelled with their new gridpoint number. Thereafter, the element/side information obtained before is employed to add the new elements. The elements to be refined are grouped together according to the refinement cases shown in Figure 14.4. Each case is treated in block fashion in a separate subroutine. This type of h-refinement owes its success to the reduction of the many possible refinement cases to only six. In order to accomplish this, some information for the 2:8+ and the 4:8+ cases is stored ahead in scratch arrays. After these elements have been refined according to the 2:8 and 4:8 cases, their sons are screened for further refinement using this information. All sons that require further refinement are then grouped together as 1:2 or 1:4 cases, and processed in turn.

As the original description of all variables was performed using linear elements, the linear interpolation of the unknowns to the new points will be conservative. However, small conservation losses will occur at curved surfaces. These losses are considered to be both unavoidable and small.

14.4.1.5. Coarsening of the grid where needed

Points and elements are deleted in two independent steps, which, in principle, could be performed in parallel.

(i) *Points*. For the points to be deleted all that remains to be done is to fill up the voids in the arrays corresponding to coordinates, unknowns and boundary conditions by renumbering points and boundary conditions.

(ii) *Elements*. The deletion of elements is again performed blockwise, by grouping together all elements corresponding to the coarsening cases shown in Figure 14.6. Thereafter, the elements are also renumbered (in order to fill up the gaps left by the deleted elements), and the point renumbering is taken into consideration within the connectivity arrays.

It is clear that the coarsening procedure is non-conservative. However, no physical or numerical problems have ever been observed by using it. This may be explained by the fact that the coarsening is done in those regions where the solution is smooth. Thus, the coarsened grid represents the solution very well, and consequently the conservation losses are small. Moreover, those regions where the maintenance of conservation is important (e.g. discontinuities) are never affected.

14.5. Examples

A number of examples are given to demonstrate some of the possible applications of adaptive mesh procedures.

14.5.1. CONVECTION BETWEEN CONCENTRIC CYLINDERS

This case, taken from van Dyke (1989), shows the use of h-refinement for buoyancy driven flows. The ratio of diameters is $d_o/d_i = 3.14$, and the Grashof number based on the gap width is $Gr = 122\,000$. The Reynolds- and Prandtl- numbers were $Re = 1.0$ and $Pr = 0.71$, respectively. After obtaining a first solution, mesh adaptation was invoked to resolve in more detail the temperature gradients. The resulting flow and temperature fields are displayed in Figure 14.8. The comparison with the results presented in van Dyke (1989) is very good. It is worth noting that the places where refinement took place are rather non-intuitive. On the other hand, the results obtained on the original, unrefined mesh did not exhibit a good comparison to the experiment.

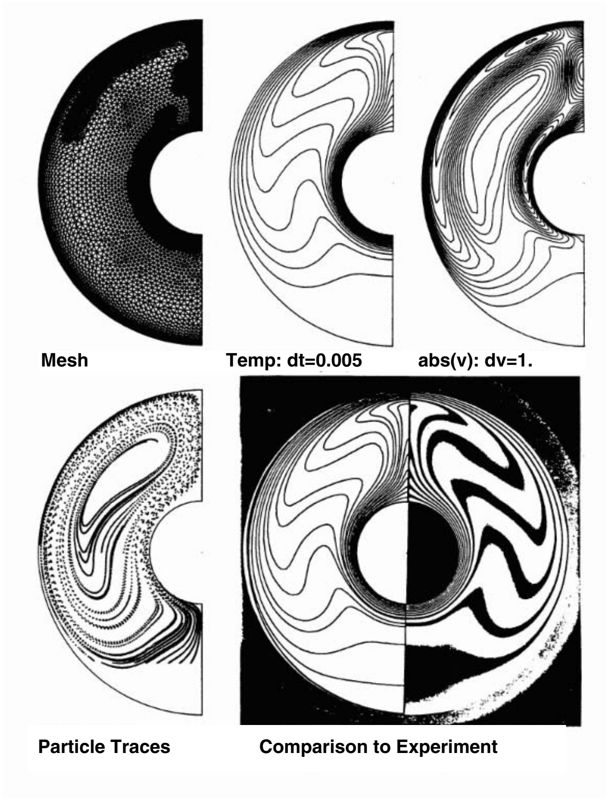
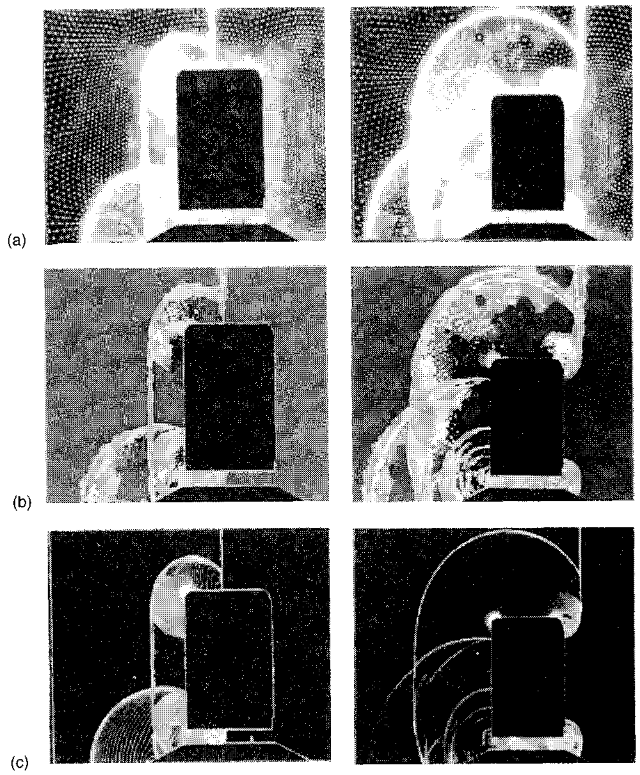
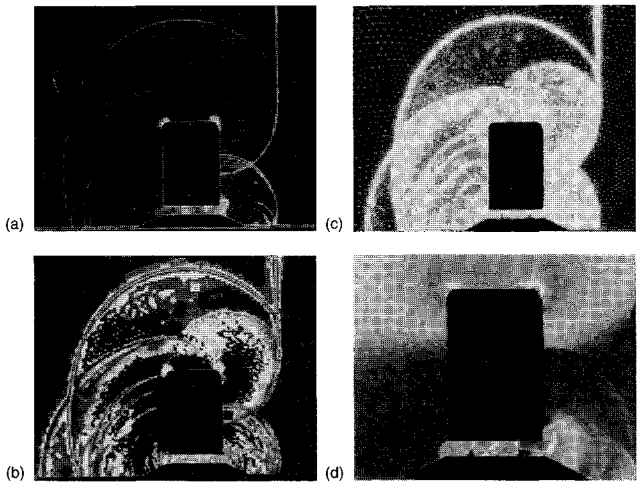


Figure 14.8. Convection between concentric cylinders

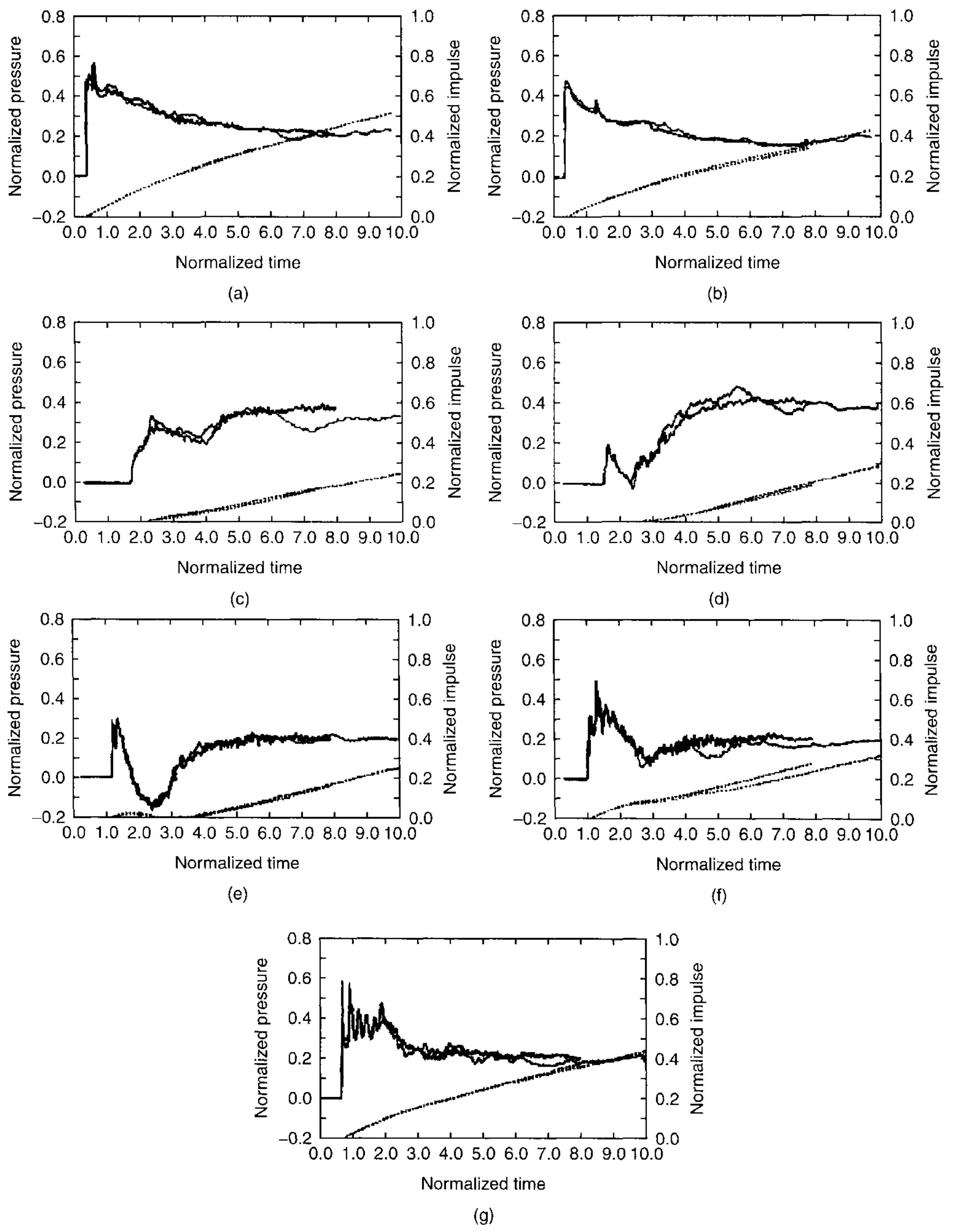


(a)



(b)

Figure 14.9. (a) and (b) Shock-object interaction; (c) comparison to experiment

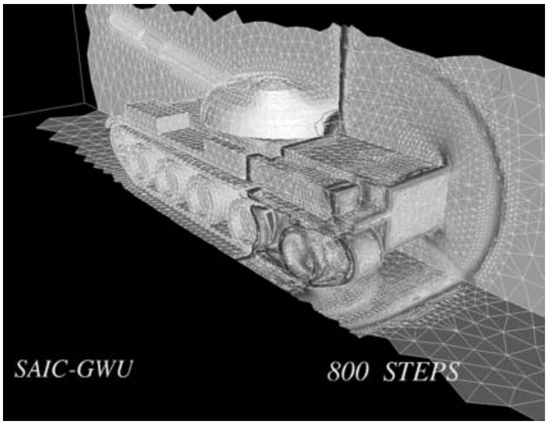


(c)

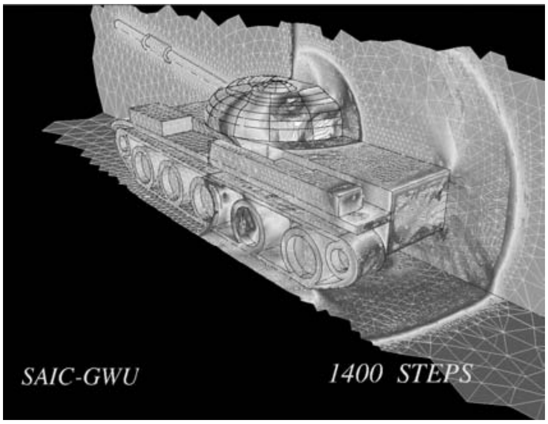
Figure 14.9. Continued

14.5.2. SHOCK-OBJECT INTERACTION IN TWO DIMENSIONS

Figures 14.9(a)–(c) show a case taken from (Baum and Löhner (1992)). They show classic h-refinement for strongly unsteady flows at its best. For this class of problems a new mesh is required every five to seven timesteps, strict conservation of mass, momentum and energy during refinement is critical, and the introduction of dissipation due to information loss during interpolation when remeshing proves disastrous for accuracy. A maximum of six levels of refinement were specified for this case, yielding meshes that on average have 300 000 triangles and 100 000 points. Figures 14.9(a) and (b) show the mesh, mesh refinement levels and pressures for different times.



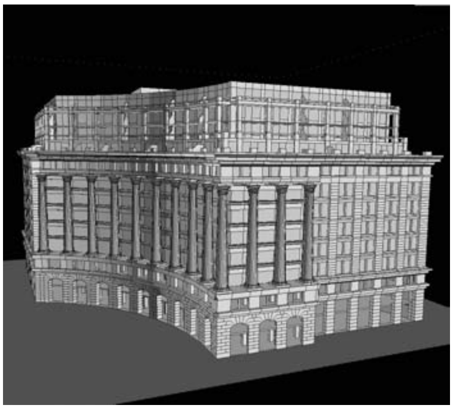
(a)



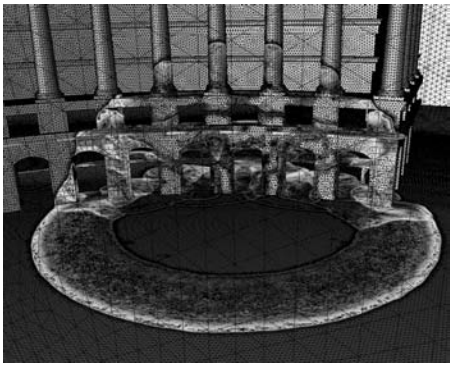
(b)

Figure 14.10. Shock–object interaction in three dimensions

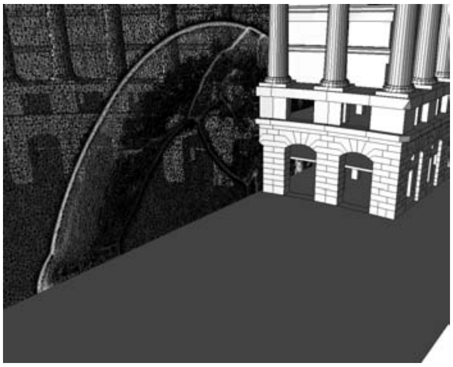
Observe the detail in the physics that is achievable through adaptation. Notice furthermore the small extent of the regions that require refinement as compared to the overall domain. The equivalent uniform mesh run would have required more than two orders of magnitude more elements, CPU time and memory, pushing the limits of available supercomputers. The



(a)



(b)



(c)

Figure 14.11. Shock–structure interaction: (a) building definition; (b) surface mesh and pressure; (c) mesh and pressure in plane

comparison to experimental results, given in Figure 14.9(c), reveals that indeed very accurate results with a minimum of degrees of freedom are achieved using adaptive grid refinement for this class of problems.

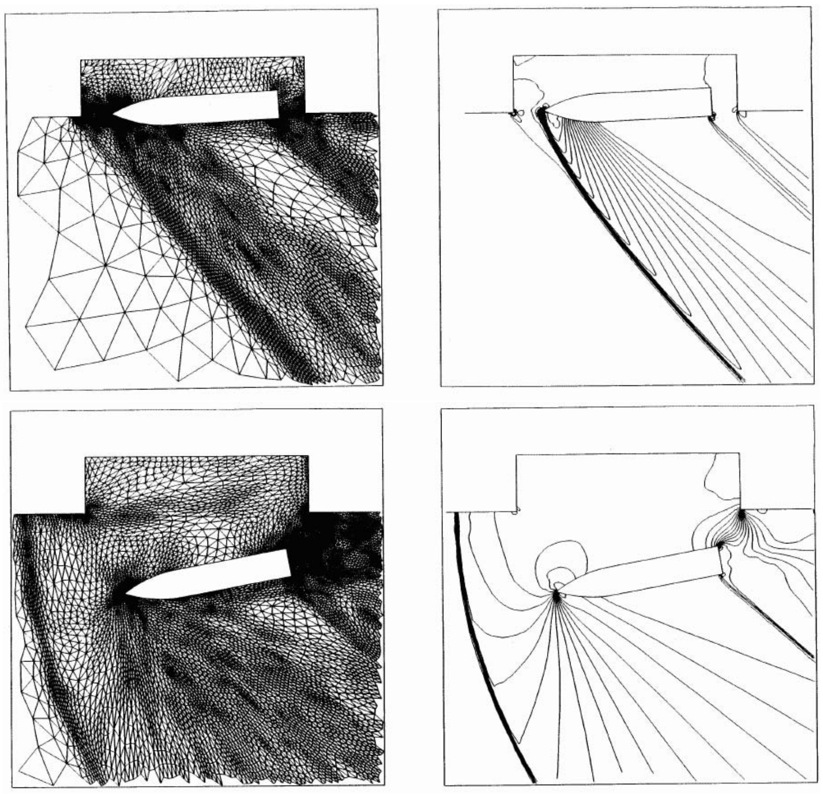


Figure 14.12. Object falling into supersonic free stream

14.5.3. SHOCK–OBJECT INTERACTION IN THREE DIMENSIONS

Figures 14.10(a)–(b) show a case taken from Baum and Löhner (1991) and Löhner and Baum (1992). The object under consideration is a common main battlefield tank. A maximum of two layers of refinement were specified close to the tank, whereas only one level of refinement was employed farther away. The original, unrefined, but strongly graded mesh consisted of approximately 100 000 tetrahedra and 20 000 points. During the run, a mesh change (refinement and coarsening) occurred every five timesteps, and the mesh size increased to approximately 1.6 million tetrahedra and 280 000 points. This represents an increase factor of 1:16. Although seemingly high, the corresponding global h-refinement would have resulted in a 1:64 size increase. A second important factor is that most of the elements of the original mesh are close to the body, where most of the refinement is going to take place. Figures 14.10(a) and (b) show surface gridding and pressure contours at two selected times during the run. The extent of mesh refinement is clearly discernable, as well as the location and interaction of shocks.

14.5.4. SHOCK–STRUCTURE INTERACTION

Figures 14.11(a)–(c) show a typical shock–structure interaction case. The building under consideration is shown in Figure 14.11(a). One layer of refinement was specified wherever the physics required it. The pressures and grids obtained at the surface and at planes at a given time are shown in Figures 14.11(b) and (c). The mesh had approximately 60 million tetrahendra.

14.5.5. OBJECT FALLING INTO SUPERSONIC FREE STREAM TWO DIMENSIONS

The problem statement is as follows: an object is placed in a cavity surrounded by a free stream at $M_\infty = 1.5$. After the steady-state solution is reached (time $T = 0.0$), a body motion is prescribed, and the resulting flowfield disturbance is computed. Adaptive remeshing was performed every 100 timesteps initially, while at later times the grid was modified every 50 timesteps. One level of global h-refinement was used to accelerate the grid regeneration. The maximum stretching ratio specified was $S = 5.0$. Figure 14.12 shows different stages during the computation at times $T = 60$ and $T = 160$. One can clearly see how the location and strength of the shocks change due to the motion of the object. Notice how the directionality of the flow features is reflected in the mesh.

15 EFFICIENT USE OF COMPUTER HARDWARE

However clever an algorithm may be, it has to run efficiently on the available computer hardware. Each type of computer, from the PC to the fastest massively parallel machine, has its own shortcomings that must be accounted for when developing both the algorithms and the simulation code. The present section assumes that the algorithm has been selected, and identifies the main issues that must be addressed in order to achieve good performance on the most common types of computers. The main types of computer platforms currently being used are as follows.

(a) *Personal computers*. Although perhaps not considered a serious analysis tool even a decade ago, personal computers can already be used cost-effectively for 3-D simulations. In fact, many applications where CPU time is not a constraining factor are currently being carried out on PCs. Most CFD software companies report higher revenues from PC platforms than from all other platforms combined. High-end PCs (4 Gbytes of RAM, 120 GFLOPS graphics card) are ideal tools for simulations. We see this as one more proof of the theme that has been repeated so often in this book: a CFD run is more than just CPU – if this were so, vector machines would have become the dominant type of computer. Rather, it consists of problem definition, grid generation, flow solver execution and visualization. High-end PCs combine a relatively fast CPU with good visualization hardware, allowing to cut down the most expensive cost-component of any run: man-hours.

(b) *Vector machines*. These machines achieve higher speeds by splitting up arithmetic operations (fetch, align, add, multiply, store, etc.), performing each on different data items concurrently. The assumption made is that the same basic operation(s) have to be performed on a relatively large number of data items. These data items can be thought of as vectors, hence the name. As an example, consider the operation `D=C*(A+B)`. While the central CPU fetches the data from memory for the ith item, it may align the data for item $i+1$, add two numbers for item $i+2$, multiply numbers for item $i+3$ and store the results for item $i+4$. This would yield a speedup of 1:4. In practice, many more operations than the ones described above are required even to add two numbers. Hence, speedups of about one order of magnitude are achievable (1:14 on the Cray-X or NEC-SX series).

(c) *Single instruction multiple data (SIMD) machines*. Here the assumption made is that all data items (e.g. elements, points, etc.) will be subject to the same arithmetic operations. In order to go beyond the one order of magnitude speedup of vector machines, thousands of processors are combined. Each processor performs the same task on a different piece of data. While this type of machine did not succeed when based on conventional chips, high-end graphics cards are increasingly being used in this mode (Hagen *et al.* (2006), LeGresley *et al.* (2007)).

(d) *Multiple instruction multiple data (MIMD) machines*. In this case different arithmetic operations may be performed on different processors. This circumvents some of the restrictions posed by the SIMD assumption that all processors are performing the same arithmetic operation. On the other hand, the operating system software required to keep these machines functioning is much more involved and sensitive than that required for SIMD machines.

The emerging architecture for future machines is a generalization of the MIMD machine, where some processors may be based on commodity, general-purpose chips, others on reduced instruction set chips (RISC-chips), others on powerful vector-processors, and some have SIMD architecture. An example of such a machine is the Cray-T3E, which combines a Cray-T90 vector supercomputer with up to 2056 Alpha-Chip-based processors. An architecture like this, which combines scalar, vector and distributed memory parallel processing, requires the programmer to take into consideration all the individual aspects encountered in each of these architectures.

15.1. Reduction of cache-misses

Indirect addressing (i/a) is required for any unstructured field solver, as different data types (point-, element-, face-, edge-based) have to be related to one another. In loops over elements or edges, data is accessed in an unordered way. Most RISC-based machines, as well as some vector machines, will store the data required by the CPU ahead of time in a cache. This cache allows the data to be fetched by the CPU much more rapidly than data stored in memory or on disk. The data is brought into the cache in the same way that it is stored in memory or on disk. If the data required by the CPU for subsequent arithmetic operations is not close enough to fit into the cache, this piece of information will have to be fetched from memory or disk. This is called a cache-miss. Depending on the frequency of cache-misses versus CPU, a serious degradation in performance, often in excess of 1:10, can take place. The relative number of cache-misses invariably increases with problem size. The aim of the renumbering strategies considered in the present section is to minimize the frequency of cache-misses, i.e. to retard the degradation of performance with problem size. The main techniques considered are:

- array access in loops;
- renumbering of points to reduce the spread in memory of the items fetched by a single element or edge;
- reordering of the nodes in each element so that data is accessed in as uniform a way as possible within each element; and
- renumbering of elements, faces and edges so that data is accessed in as uniform a way as possible when looping over them.

15.1.1. ARRAY ACCESS IN LOOPS

Storing all the arrays required (elements, coordinates, unknowns, edges, etc.) in a way that is compatible with the way they are accessed within loops reduces cache-misses appreciably. To see why, consider the array containing the coordinates of the points: horizontal or flat storage à la `coord(ndimn,npoin)` would be the preferred choice for a workstation, whereas for some Crays the preferred choice would be vertical storage à la `coord(npoin,ndimn)`.

Suppose that the difference vector $(dx,\ dy,\ dz)$ of the two endpoints of an edge is required. This implies fetching six items and performing three arithmetic operations. For flat storage, the jump in memory is given by

```
Get x1=coord(1,ipoi1)          Jump to ipoi1
Get y1=coord(2,ipoi1)          Jump by 1
Get z1=coord(3,ipoi1)          Jump by 1
Get x2=coord(1,ipoi2)          Jump to ipoi2
Get y2=coord(2,ipoi2)          Jump by 1
Get z2=coord(3,ipoi2)          Jump by 1
```

whereas for vertical storage the jumps are

```
Get x1=coord(ipoi1,1)          Jump to ipoi1
Get x2=coord(ipoi2,1)          Jump to ipoi2
Get y1=coord(ipoi1,2)          Jump by mpoin
Get y2=coord(ipoi2,2)          Jump to ipoi2
Get z1=coord(ipoi1,3)          Jump by mpoin
Get z2=coord(ipoi2,3)          Jump to ipoi2
```

The difference in the number of large jumps is clearly visible from this comparison. For this reason, flat storage is recommended for any machine with cache. Note that, for codes written in C, the opposite holds, as the second index moves faster than the first one.

15.1.2. POINT RENUMBERING

Consider the evaluation of an edge RHS (the same basic principle applies to element-based or face-based solvers), given by the following loop.

```
Loop 1:

      do 1600 iedge=1,nedge
      ipoi1=lnoed(1,iedge)
      ipoi2=lnoed(2,iedge)
      redge=geoed(  iedge)*(unkno(ipoi2)-unkno(ipoi1))
      rhspo(ipoi1)=rhspo(ipoi1)+redge
      rhspo(ipoi2)=rhspo(ipoi2)-redge
 1600 continue
```

The operations to be performed can be grouped into the following three major steps (see Chapter 8):

(a) gather point information into the edge;

(b) perform the required mathematical operations at edge level;

(c) scatter-add the edge RHS to the assembled point RHS.

The transfer of information to and from memory required in steps (a), (c) is proportional to the number of nodes in the edge (element, face) and the number of unknowns per node. If the nodes within each edge (element, face) are widely spaced in memory, cache-misses are likely to occur. If, on the other hand, all the points within an element are 'close' in memory, cache-misses are minimized. From these considerations, it becomes clear that cache-misses are directly linked to the bandwidth of the equivalent matrix system (or graph). Point renumbering to reduce bandwidths has been an important theme for many years in traditional finite element applications (Piessanetzky (1984), Zienkiewicz (1991)). The aim was to reduce the cost of the matrix inversion, which was considered to be the most expensive part of any finite element simulation.

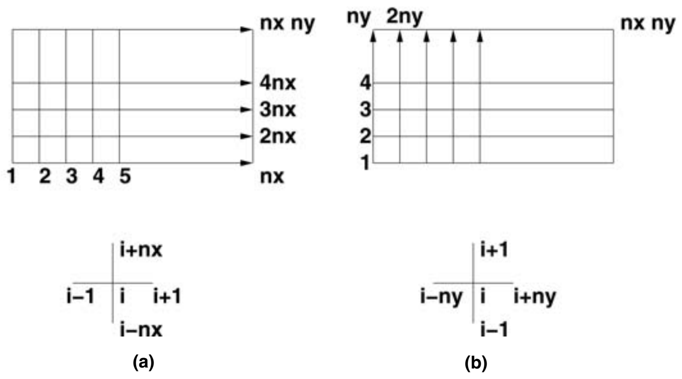


Figure 15.1. Ordering of points for 2-D mesh

The optimal renumbering of points in such a way that spatial (or near-neighbour) locality is mirrored in memory is a problem of formidable algorithmic complexity. Fortunately, most of the benefits of renumbering points are already obtained from near-optimal heuristic renumbering techniques. To see how most of these fast, near-optimal techniques work, consider the rectangular domain with a structured mesh shown in Figure 15.1. Numbering the points in the horizontal (Figure 15.1(a)) and vertical (Figure 15.1(b)) directions yields an average bandwidth of nx and ny, respectively. One should therefore aim to number the points in the direction normal to the longest graph depth. Based on this observation, several point renumbering techniques have been developed. To exemplify these techniques, the simple mesh shown in Figure 15.2 is considered.

15.1.2.1. Directional ordering

If the direction of maximal graph depth is known, one can simply order the points in this direction. This is perhaps the simplest (and fastest) possible renumbering, but implies that the problem class being addressed has a clear maximal graph depth direction that can easily be identified. Renumbering in the x-direction, this yields the numbering shown in Figure 15.3.

15.1.2.2. Bin ordering

Given an arbitrary distribution of points, one may first place the points in a bin of uniform size h. One can then identify, by ordering the number of bins in the x, y, z directions in

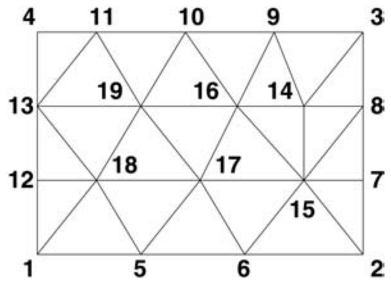


Figure 15.2. Original mesh

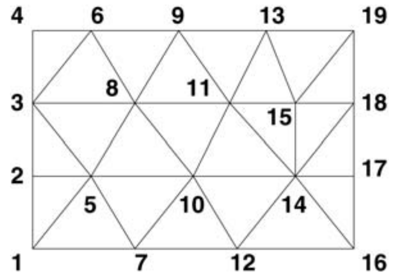


Figure 15.3. Renumbering in the x-direction

ascending size `i,j,k`, the plane k that traverses space yielding the lowest bandwidth, i.e. the closest proximity in memory. Bins offer the advantage of high speed (very few operations are required, and most of these are easy to vectorize/ parallelize) and simplicity. After obtaining the overall dimensions of the computational domain, bin ordering may be realized in two ways:

(1) Obtain the bin each point falls into; store the points into bins (e.g. using a linked list via `lbin1(1:npoin),lbin2(1:npoin+1)`); go through the bins, renumbering points;

(2) Obtain the bin each point falls into; assign a number to the point based on the bin it falls into (e.g. `inumb=ibinx+nbinx*(ibiny-1)+nbinx*nbiny*(ibinz-1)`); store the points in a heap list (based on the assigned number); retrieve the points from the heap list, renumbering points.

Bins are mostly used for grids with modest changes in element size. Figure 15.4 shows the bin ordering of points for the mesh from Figure 15.2.

15.1.2.3. *Quad/octree ordering*

For grids that exhibit large variations in element size, the bin ordering described above will yield suboptimal renumberings, as some bins will have many points that could be ordered in a better way. A way to circumvent this shortcoming is to allow for bins of variable size, i.e. allowing only a certain number of points within each bin (or regular region of

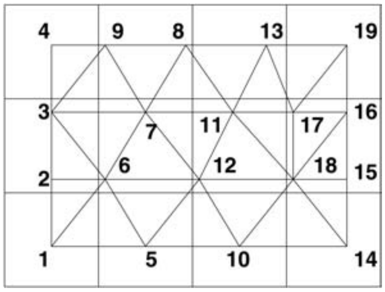


Figure 15.4. Renumbering using bins

subdivided space). This is easily accomplished using quadtrees (two dimensions) or octrees (three dimensions). These data structures have already been described in Chapter 2. Having stored all the points, the quad/octree is traversed as shown in Figure 15.5, renumbering the points. One can see that in this way spatial proximity is mirrored in memory in a near-optimal way.

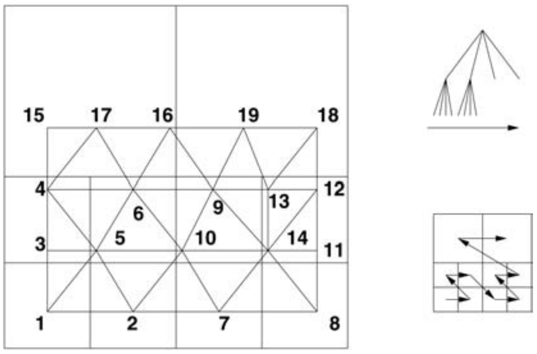


Figure 15.5. Renumbering using quadtree

15.1.2.4. *Space-filling curves*

A very similar effect to that of quad/octree ordering can be achieved by using so-called space-filling curves. A typical curve that is often employed is the Peano–Hilbert–Morton curve shown in Figure 15.6 for two dimensions. Any point in space can be thought of as lying on this curve. This implies that, once the coordinate along this line ξ has been established for each point, the points can be renumbered in ascending order of ξ. One can see from Figure 15.6 the similarity with quad/octree renumbering, as well as the effectiveness of the procedure.

15.1.2.5. *Wave renumbering*

All of the techniques discussed so far have only required the spatial location of points to achieve near-optimal renumberings. However, if a mesh is given, one can obtain from

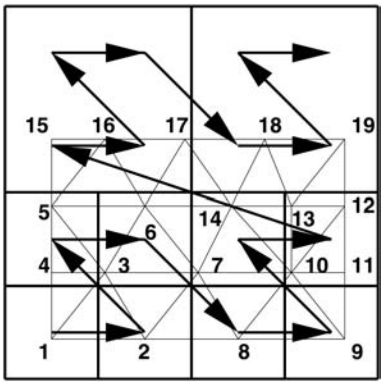


Figure 15.6. Renumbering using space-filling curves

the connectivity table the nearest-neighbours for each point and construct renumbering techniques with this information. One of the most useful techniques is the so-called wave renumbering or advancing-front renumbering technique. Starting from a given point, new points are added in layers according to the smallest connectivity. The 'front' of renumbered points is advanced through the grid until all points have been covered (see Figure 15.7).

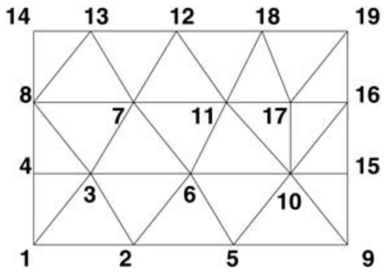


Figure 15.7. Wave front renumbering

The choice of the seed-point can have a significant effect on the total bandwidth obtained. Unfortunately, choosing the optimal starting point may be more expensive than the whole subsequent simulation. A very effective heuristic approach (all of the bandwidth minimization strategies are heuristic by nature) is to choose the last point of the renumbered mesh as the starting point for a new renumbering pass. This procedure is repeated until no further reduction in the bandwidth is achieved. Convergence is obtained in a relatively small number of passes, typically less than five, even for complex 3-D meshes.

An improvement on the wave renumbering technique is the Cuthill–McKee (Cuthill and McKee 1969) or reverse Cuthill–McKee (RCM) reordering. At each stage, the node with the smallest number of surrounding unrenumbered nodes is added to the renumbering table. For meshes, which are characterized by having a bounded number of nearest-neighbours for each point, the improvement of RCM versus wave front is not considerable.

15.1.3. REORDERING OF NODES WITHIN ELEMENTS

Cache-misses at the element level may be reduced further by reordering the nodes in each element. The idea is to access the required point information for each element in as uniform a way as possible. This may be seen from the following example. Consider a tetrahedral element with nodes

```
inpoel(1:4,ielem)= 100, 5000, 400, 4600.
```

Clearly a better way to define this element would be

```
inpoel(1:4,ielem)= 100, 400, 4600, 5000.
```

Observe that the element definition (volume, shape functions, etc.) remains the same as before. However, as the nodal information is accessed, the two large ‘jumps’ from point `ipoi1=100` to `ipoi2=5000` and from `ipoi3=400` to `ipoi4=4600` have been reduced to one ‘jump’ from `ipoi3=400` to `ipoi4=4600`, i.e. potential cache-misses have been halved. An example of a possible table-based algorithm that accomplishes such a reordering of nodes for tetrahedra is the following:

RE1. Define the switching tables `lnod2(4,4)`, `lnod3(4,4)` as

```
data lnod2/ 0 , 3 , 4 , 2 ,
            4 , 0 , 1 , 3 ,
            2 , 4 , 0 , 1 ,
            3 , 1 , 2 , 0 /
data lnod3/ 0 , 4 , 2 , 3 ,
            3 , 0 , 4 , 1 ,
            4 , 1 , 0 , 2 ,
            2 , 3 , 1 , 0 /
```

RE2. Loop over the elements, reordering the nodes:

- Get the nodes `inmin`, `inmax` corresponding to the minimum and maximum point in the element;
- The new element definition is given by:

```
ipoi1=inpoel(            inmin ,ielem)
ipoi2=inpoel(lnod2(inmax,inmin),ielem)
ipoi3=inpoel(lnod3(inmax,inmin),ielem)
ipoi4=inpoel(            inmax ,ielem)
```

Obviously, for other types of elements other local reordering strategies are possible.

15.1.4. RENUMBERING OF EDGES ACCORDING TO POINTS

When possible cache-misses in each edge have been minimized, the next step is to minimize cache-misses as different edges are processed. In an ideal setting, as the loop over the edges progresses the data accessed from and sent to point arrays should increase as monotonically as possible. This would preserve data locality as much as possible. The techniques discussed in the following for edges may be applied for element or face loops in the same manner.

Assume a mesh of `nedge` edges and `npoin` points. A simple algorithm that accomplishes such a renumbering is the following (see Figure 15.8):

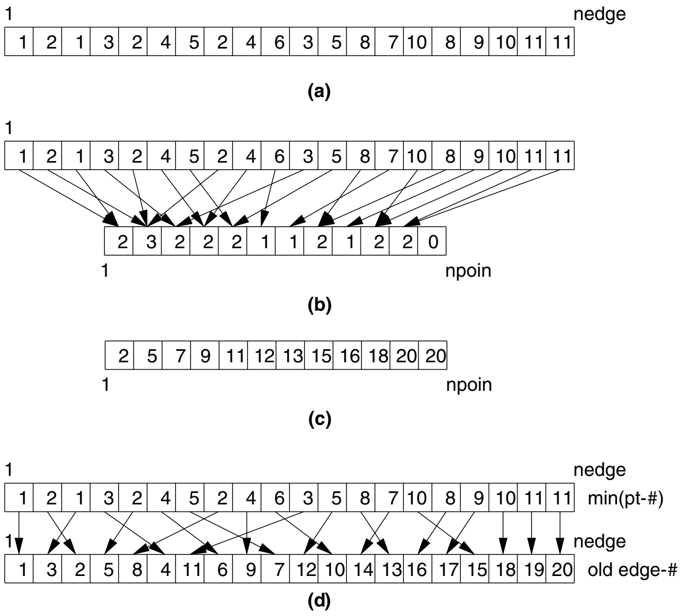


Figure 15.8. Renumbering of edges according to points: (a) minimum point number in each edge; (b) assessment of storage location; (c) reordering of storage location; (d) reordering of edges

R1. Compute the minimum point touching each edge and store it in `lemin(1:nedge)`;

R2. Initialize the pointer array `lpoin(1:npoin)=0`;

R3. Loop over the edge, storing in `lpoin` the number of times the minimum point in each edge is accessed;

R4. Add all the entries in `lpoin` in order to obtain the storage locations for the edge reordering;

R5. Loop over the edges:

```
Obtain the new edge location from ipmin=lemin(ielem) and
ienew=lpoin(ipmin);
Store the new edge location: lenew(ienew)=iedge;
Update the storage location in lpoin: lpoin(ipmin)=ienew+1;
```

R6. Renumber the edges with `lenew(1:nelem)`.

The renumbering described above will only order the edges in clusters according to their minimum point. A *two-pass strategy* yields a more uniform edge-to-point renumbering. In the first pass, the edges are ordered according to their *maximum* point number using the algorithm outlined above. In the second pass, the edges are again renumbered, but this time ordered according to their *minimum* point number. Because the edge order for points with the same minimum point number is left untouched by the renumbering algorithm, a uniform min/max ordering of edges is obtained.

Table 15.1. Timings for the train problem (240 000 tetrahedra, 48 000 points, 340 000 edges, IBM-6000/530)

`frntmin`	`renuemi`	`ordnoel`	`renuel`	CPU (s)
OFF	OFF	OFF	`mvmax`	958
OFF	ON	OFF	`mvmax`	882
ON	ON	OFF	`mvmax`	834
OFF	ON	OFF	`128`	597
OFF	ON	OFF	`nelem`	534
ON	ON	OFF	`nelem`	524
ON	ON	ON	`nelem`	510

15.1.5. SOME TIMINGS

All of the algorithms described for the reduction of cache-misses are of linear $O(N)$ complexity and simple to code. The following example should suffice to demonstrate their effectiveness. The code used was FEFLO93, an explicit edge-based arbitrary Lagrangian–Eulerian (ALE) finite element Euler solver developed by the author. The basic CFD solver employs a Galerkin finite element discretization of the fluxes, as well as a blend of second- and fourth-order damping operators. This leads to simple loops, with a high ratio of i/a:FLOPS (floating point operations). The mesh consisted of about 45 000 points and 240 000 tetrahedra. A large portion of the mesh was moving, requiring the re-evaluation of shape-function derivatives, volumes and other geometrical parameters at each timestep. The code was exercised for 20 timesteps, which is sufficient to reduce the start-up cost of the run to less than 10%. Table 15.1 lists the performances observed with the different renumbering options. `frntmin` denotes the advancing front point renumbering, `renuemi` the renumbering of elements according to smallest point number, `renuel` the grouping of elements for vectorization and `ordnoel` the internal reordering of nodes within each element. `frntmin` achieved an average bandwidth reduction in excess of 1:10 for this mesh. This is consistent with most of the grids of this size generated using the advancing front grid generator. The workstation used had 64 Mbytes of memory. The first observation is that although the number of FLOPS was identical, the observed CPU requirements varied by a factor of two. This implies that cache-misses indeed play a significant role in degrading performance. The second observation is that the renumbering used for the Crays (i.e. using the maximum possible vector length `mvecl=mvmax`, typical of production codes) gave the worst performance. The effect of this renumbering is far more important than even point renumbering. The cache-misses occur because each element/edge group traverses the complete point database. This amounts to $O(25)$ traversals. Local reuse of data with vectorizable loops of length `mvlen=128` gave timings that were close to the ones observed without renumbering.

In order to assess the effect of the described renumbering strategies for a Cray, a similar series of tests was conducted on a Cray-YMP. The results obtained have been compiled in Table 15.2.

As one can see, the performance on the Cray-YMP is not affected by the renumbering strategies discussed above. This is to be expected, as this is a fixed-memory machine and the

Table 15.2. Timings for the box problem (25 000 tetrahedra, 6000 points, 35 000 edges), Cray-YMP

`frntmin`	`renuemi`	`ordnoel`	`renuel`	CPU (s)
OFF	ON	ON	`mvmax`	55.0
ON	ON	ON	`mvmax`	54.7
OFF	ON	ON	`128`	54.2
ON	ON	ON	`128`	55.2

number of edges was $64 \times O(500)$, so that the performance degradation due to short groups is minimal.

15.1.6. AGGLOMERATION TECHNIQUES

Inspection of Loop 1 (see above) indicates that it requires six indirect addressing (i/a) operations and four FLOPs per edge. While many vendors quote impressive megaflops (10^6 FLOPs/s) or gigaflops (10^9 FLOPs/s) for their chips, production codes seldomly achieve even 10% of these peak numbers. The reason is that these chips only achieve peak performance for situations where a large number of operations is carried out for each item transferred from and to memory. As one can see, Loop 1, like most loops encountered in CFD codes, does not fall into this category. Given that i/a accounts for a large portion of total CPU cost, alternative data structures that reduce i/a even further will have a considerable pay-off. Three related possibilities that achieve this goal: stars, chains and superedges, are examined in the following. All of these possibilities make the assumption that, once the data has been gathered from memory, it may reside and be reused at negligible cost in register space. Thus, the design criteria is to operate on gathered data as much as possible.

15.1.6.1. Stars

A first way to reduce i/a is to split the main loop over the edges into a loop over the points and a loop over the edges surrounding each point. Such a breakup may look like the following:

```
      do 1000 ipoi1=1,npoin
      iedg0=esup2(ipoi1)+1
      iedg1=esup2(ipoi1+1)
      upoi1=unkno(ipoi1)
       do 1200 iedge=iedg0,iedg1
       ipoi2=esup1(iedge)
       redge=edlap(iedge)*(upoi1-unkno(ipoi2))
       rhspo(ipoi1)=rhspo(ipoi1)+redge
       rhspo(ipoi2)=rhspo(ipoi2)-redge
1200  continue
1000 continue
```

This loop requires a total of 3 i/a operations per edge. This represents a saving factor of 2:1 in i/a operations. The original `lpoed(2,iedge)` data structure could have been maintained, but going to a pointer structure almost halves the required storage. The main disadvantage of this double loop is that the inner loop is very short. Typical numbers are three for triangles and seven for tetrahedra. Thus, it will not run efficiently on vector machines. This can be remedied by inverting the loops, leading to a point-based data structure which is denoted as *star*.

As the grid is assumed to be unstructured, several different star types will have to be used in order to cover all edges. The family of possible stars is shown in Figure 15.9. A typical loop for 6-stars, using the conventional `lpoed(2,nedge)` data structure for simplicity, may look as follows:

```
      do 1000 iedge=iedg0,iedg1
      ipoi0=lpoed(1,iedge  )
      ipoi1=lpoed(2,iedge  )
      ipoi2=lpoed(2,iedge+1)
      ipoi3=lpoed(2,iedge+2)
      ipoi4=lpoed(2,iedge+3)
      ipoi5=lpoed(2,iedge+4)
      ipoi6=lpoed(2,iedge+5)
      upoi0=unkno(ipoi0)
      redg1=edlap(iedge  )*(upoi0-unkno(ipoi1))
      redg2=edlap(iedge+1)*(upoi0-unkno(ipoi2))
      redg3=edlap(iedge+2)*(upoi0-unkno(ipoi3))
      redg4=edlap(iedge+3)*(upoi0-unkno(ipoi4))
      redg5=edlap(iedge+4)*(upoi0-unkno(ipoi5))
      redg6=edlap(iedge+5)*(upoi0-unkno(ipoi6))
      rhspo(ipoi0)=redg1+redg2+redg3+redg4+redg5+redg6
      rhspo(ipoi1)=rhspo(ipoi1)-redg1
      rhspo(ipoi2)=rhspo(ipoi2)-redg2
      rhspo(ipoi3)=rhspo(ipoi3)-redg3
      rhspo(ipoi4)=rhspo(ipoi4)-redg4
      rhspo(ipoi5)=rhspo(ipoi5)-redg5
      rhspo(ipoi6)=rhspo(ipoi6)-redg6
 1000 continue
```

Table 15.3 lists other possibilities, as well as the saving factors in i/a that can be achieved.

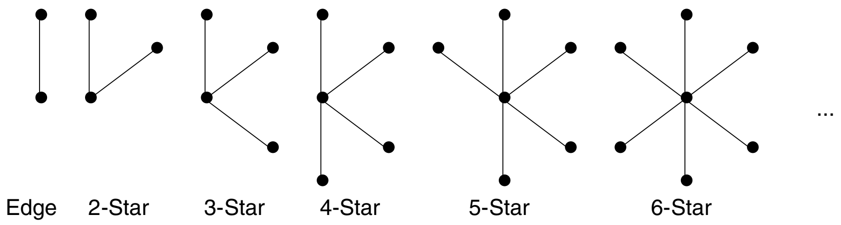


Figure 15.9. Agglomeration of edges into stars

These reduction ratios may be improved if the points are renumbered according to the family of stars, thereby avoiding the i/a of `ipoi0`-related arrays. However, such a specialized

Table 15.3. Family of stars

Type	Edges	Points	i/a/edge	i/a reduction
Edge	1	2	6:1	1.00
2-star	2	3	9:2	1.33
3-star	3	4	4:1	1.50
4-star	4	5	15:4	1.60
5-star	5	6	18:5	1.67
6-star	6	7	7:2	1.71

point renumbering may lead to new problems, such as large bandwidths and cache-misses. Therefore, the more conservative, but realistic, reduction factors achievable for any mesh are quoted here. For a general mesh, the total reduction in i/a will depend on the ratio of ‘large’ stars to ‘small’ stars. One can see, however, that the asymptotic factor is in the neighbourhood of 1:1.70. This is not very impressive, and certainly not worth a major code change.

15.1.6.2. Superedges

The second way to reduce i/a is the use of ordered groups of edges, which are denoted as *superedges*. A superedge loop with count six for tetrahedra looks like the following.

```
      do 1000 iedge=iedg0,iedg1,6
      ipoi1=lpoed(1,iedge  )
      ipoi2=lpoed(2,iedge  )
      ipoi3=lpoed(1,iedge+3)
      ipoi4=lpoed(2,iedge+3)
      upoi1=unkno(ipoi1)
      upoi2=unkno(ipoi2)
      upoi3=unkno(ipoi3)
      upoi4=unkno(ipoi4)
      redg1=edlap(iedge  )*(upoi2-upoi1)
      redg2=edlap(iedge+1)*(upoi3-upoi2)
      redg3=edlap(iedge+2)*(upoi3-upoi1)
      redg4=edlap(iedge+3)*(upoi4-upoi1)
      redg5=edlap(iedge+4)*(upoi4-upoi2)
      redg6=edlap(iedge+5)*(upoi4-upoi3)
      rhspo(ipoi1)=rhspo(ipoi1)+redg1+redg3+redg4
      rhspo(ipoi2)=rhspo(ipoi2)-redg1+redg2+redg5
      rhspo(ipoi3)=rhspo(ipoi3)-redg2-redg3+redg6
      rhspo(ipoi4)=rhspo(ipoi4)-redg4-redg5-redg6
 1000 continue
```

This loop requires 12 i/a operations for every six edges. This represents a saving factor of 3:1 in i/a operations, and a considerable improvement over stars. The superedge consisting of six linked edges is just one of a family of possible superedges. Table 15.4 lists other possibilities, as well as the saving factors in i/a that can be achieved.

Table 15.4. Family of superedges

Type	Edges	Points	i/a/edge	i/a reduction
Edge	1	2	6:1	1.00
V-Edges	2	3	9:1	1.33
Triangle	3	3	3:1	2.00
DoubleTria	5	4	12:5	2.50
QuadruTria	9	6	2:1	3.00
Tetrahedron	6	4	2:1	3.00
DoubleTetra	9	5	15:9	3.60
TripleTetra	12	6	3:2	4.00
QuintuTetra	15	7	21:15	4.28

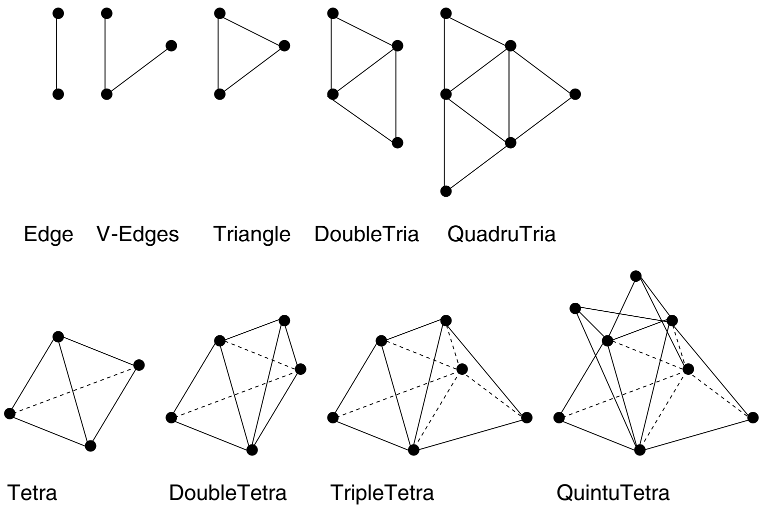


Figure 15.10. Superedges

These superedges have been illustrated in Figure 15.10. It is easy to see that, by increasing the number of closely linked edges, the i/a:edge ratio can be lowered to an asymptotic value of 1:1. This represents a saving factor of 1:6 over conventional edges. On the other hand, as seen from the superedge-loop above, the larger the number of edges in a group, the longer and more complicated the loops. At some stage register availability will also become a constraining factor. Therefore, a balance has to be struck between efficiency and code simplicity, clarity and maintenance. For tetrahedral meshes, the most natural superedge is the tetrahedron itself. With a simple renumbering strategy, around 70% of all edges were covered with this type of superedge. Around 20% of the edges were grouped into triangles, the next most efficient superedge, and the remaining edges were treated as usual. This partitioning of edges seemed to be constant for a variety of different meshes. The saving factor in i/a for such a partitioning of edges is

$$S = \frac{i/a(\text{superedge})}{i/a(\text{usualedge})} = 0.7/3 + 0.2/2 + 0.1 = 2.3.$$

15.1.6.3. Chains

The third way to reduce i/a is to reuse old data inside a long loop. As with superedges, groups of edges are processed at the same time. However, the last point is reused with the assumption `lpoed(2,iedge)=lpoed(1,iedge+1)`. This assumption will not be valid throughout the complete mesh, but sufficiently long loops may be obtained for high i/a reductions. For the simplest possible chain one obtains

```
      ipoi1=lpoed(1,iedg0)
      upoi1=unkno(ipoi1)

      do 1000 iedge=iedg0,iedg1
      upoi0=upoi1
      ipoi1=lpoed(2,iedge)
      upoi1=unkno(ipoi1)
      redge=edlap(iedge)*(upoi0-upoi1)
      rhspo(ipoi0)=rpoi1+redge
      rpoi1=-redge
1000 continue

      rhspo(ipoi1)=rpoi1
```

Tables 15.5–15.7 list other possible chains, which also assume that more than one point is transferred between elements of the chain, as well as the saving factors in i/a that can be achieved. These chains have been illustrated in Figure 15.11.

Table 15.5. Family of 1-chains

Type	Edges	Points	i/a/edge	i/a reduction
Edge	1	2	6:1	1.00
Edge-Chain	1	1	3:1	2.00
Triangle	3	2	2:1	3.00
DoubleTria	5	3	9:5	3.33
Tetrahedron	6	3	3:2	4.00
DoubleTetra	9	4	4:3	4.50
QuintuTetra	12	5	5:4	4.80

Table 15.6. Family of 2-chains

Type	Edges	Points	i/a/edge	i/a reduction
Edge	1	2	6:1	1.00
Triangle	2	1	3:2	4.00
Tetrahedron	5	2	6:5	5.00

Table 15.7. Family of 3-chains

Type	Edges	Points	i/a/edge	i/a reduction
Edge	1	2	6:1	1.00
Tetrahedron	3	1	1:1	6.00

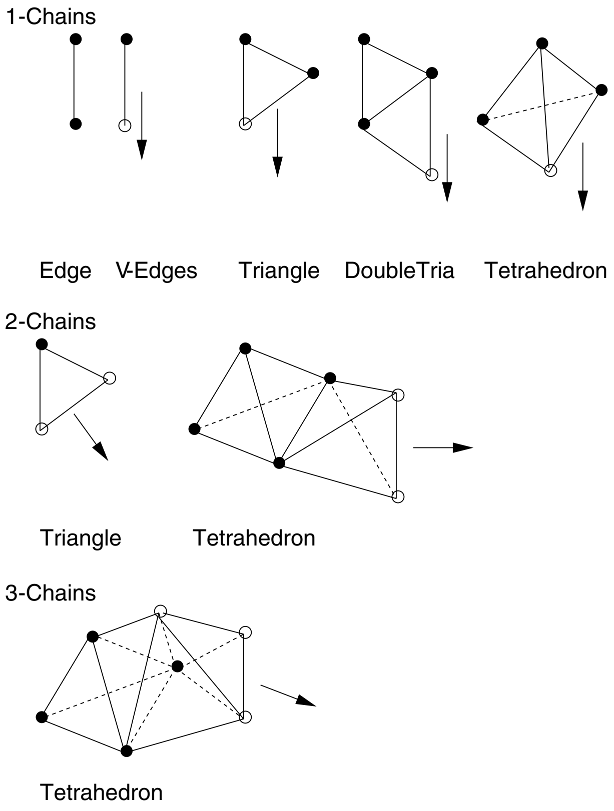


Figure 15.11. Agglomeration of edges into chains

15.1.6.4. Timings

Timing studies on several computers were conducted for the Laplacian-like loop shown above as Loop 1. Two meshes with `npoin=1,000` points and `npoin=10,000` points respectively were assumed. Furthermore, to simplify matters, the number of edges was set to `nedge=6*npoin`. The loops were performed 1000 and 100 times, respectively, for the meshes. Two cases were considered. The first was the Laplacian of a scalar, characteristic for heat conduction (Gaski and Collins (1987), Owen *et al.* (1990)). In the second case the Laplacian operator was applied to a vector quantity of five variables, representative of dissipation operators for the compressible Euler equations (see Chapter 8

Table 15.8. Timings for the scalar Laplacian

Platform	`npoin`	Edge type	CPU	Speedup
IBM-6000/530	1000	Edge	6.90	1.00
IBM-6000/530	1000	Triangle	3.65	1.89
IBM-6000/530	1000	Tetrahedron	2.92	2.36
IBM-6000/530	1000	2-ChainTetra	2.63	2.62
IBM-6000/530	10 000	Edge	13.4	1.00
IBM-6000/530	10 000	Triangle	7.21	1.85
IBM-6000/530	10 000	Tetrahedron	5.26	2.54
IBM-6000/530	10 000	2-ChainTetra	3.83	3.50
Cray-YMP	10 000	Edge	4.92	1.00
Cray-YMP	10 000	Triangle	3.13	1.57
Cray-YMP	10 000	Tetrahedron	2.60	1.89

Table 15.9. Timings for the vector Laplacian

Platform	`npoin`	Edge type	CPU	Speedup
IBM-6000/530	1000	Edge	15.66	1.00
IBM-6000/530	1000	Triangle	11.90	1.32
IBM-6000/530	1000	Tetrahedron	11.55	1.36
IBM-6000/530	1000	2-ChainTetra	11.30	1.39
IBM-6000/530	10 000	Edge	35.88	1.00
IBM-6000/530	10 000	Triangle	20.14	1.78
IBM-6000/530	10 000	Tetrahedron	17.43	2.06
IBM-6000/530	10 000	2-ChainTetra	14.13	2.54
Cray-YMP	10 000	Edge	1.89	1.00
Cray-YMP	10 000	Triangle	1.25	1.51
Cray-YMP	10 000	Tetrahedron	1.16	1.63

and Jameson (1983), Mavriplis and Jameson (1990), Mavriplis (1991b), Peraire *et al.* (1992a), Weatherill *et al.* (1993c), Luo *et al.* (1993)), and hourglass control operators used in CSD codes (Belytchko and Hughes (1983), Haug *et al.* (1991)). Because only the superedge grouping is vectorizable without a major re-write of current edge-based codes, only this type of i/a reduction technique was considered for the Cray-YMP. The results obtained have been compiled in Tables 15.8 and 15.9. Observe that although the number of FLOPS was kept identical for all cases, the superedge and chain loops achieve a considerable speedup as compared to the standard loop that is used in most current codes. As expected, the chained tetrahedron performs best on the workstation, and the tetrahedron performs well on both the workstation and the Cray-YMP. It is also interesting to note that for the vector Laplacian the triangle performs almost as well as the tetrahedron. We attribute this unexpected result to the high number of required register locations for the tetrahedron superedge loop for the case of the vector Laplacian. For the workstation, the superedge and chain loops reduce cache-misses. This is reflected in the superior speedups observed when switching to the larger mesh. For further timings of edge-based solvers for solid mechanics and transport problems, see Martins *et al.* (2000), Coutinho *et al.* (2001) and Souza *et al.* (2005).

15.2. Vector machines

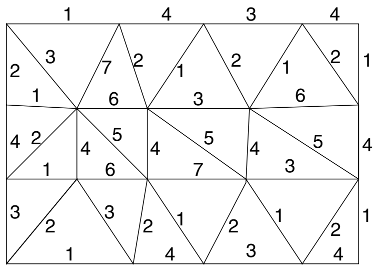
At the beginning of the 1980s, two different classes of vector machines were in use: memory-to-memory (Texas Instruments ASC, CDC Cyber-205) and register-to-register (Cray, NEC, Fujitsu, Hitachi) architectures. Owing to its greater versatility, the second type of machine is the dominant vector machine at present. Register-to-register machines, like RISC-based machines, prefer chunky loops, high flop-to-memory access ratios, and little indirect addressing. Unfortunately, most simple flow solvers look just the opposite: they have extremely simple loops, low flop-to-memory access ratios and abundant indirect addressing. Just to illustrate the importance of indirect addressing, the reader may be reminded that, even with hardware gather/scatter, it takes the equivalent of 2.5 multiplications to get a number from memory using indirect addressing. Therefore, an important issue when coding for performance on these machines is the reduction of indirect addressing operations required. For tetrahedral elements, the number of indirect addressing operations required can be approximately halved by going from an element-based data structure to an edge-based data structure. This change of data structure avoids redundant information, also leading to a proportional reduction in CPU and memory requirements (see Chapter 10). A second important issue is the vectorizability of scatter-add loops. Consider again the following edge RHS assembly loop.

```
Loop 1:
      do 1600 iedge=1,nedge
      ipoi1=lnoed(1,iedge)
      ipoi2=lnoed(2,iedge)
      redge=geoed(  iedge)*(unkno(ipoi2)-unkno(ipoi1))
      rhspo(ipoi1)=rhspo(ipoi1)+redge
      rhspo(ipoi2)=rhspo(ipoi2)-redge
1600 continue
```

The possibility of a memory overlap that would occur if two of the nodes within the same group of edges being processed by the vector registers are the same inhibits vectorization. If the edges can be reordered into groups such that within each group none of the nodes are accessed more than once, vectorization can be enforced using compiler directives. Denoting the grouping array by `edpas`, the vectorized RHS assembly would look like the following.

```
Loop 2:
        do 1400 ipass=1,npass
        nedg0=edpas(ipass)+1
        nedg1=edpas(ipass+1)
cdir$ ivdep                              ! Cray Ignore Vector DEPendencies
        do 1600 iedge=nedg0,nedg1
        ipoi1=lnoed(1,iedge)
        ipoi2=lnoed(2,iedge)
        redge=geoed(  iedge)*(unkno(ipoi2)-unkno(ipoi1))
        rhspo(ipoi1)=rhspo(ipoi1)+redge
        rhspo(ipoi2)=rhspo(ipoi2)-redge
 1600  continue
 1400 continue
```

Algorithms that group the edges into non-conflicting groups fall into the category of colouring schemes. This has been exemplified for the 2-D mesh shown in Figure 15.12.



Group	#Edges
1	10
2	9
3	7
4	9
5	3
6	3
7	2

Figure 15.12. Renumbering of edges for vectorization

The techniques discussed below consider:

- reordering the edges into groups so that point data is accessed once only in each edge group;
- balancing of groups with a switching algorithm;
- a combination of renumbering techniques to simultaneously achieve vectorizability and minimization of cache-misses.

15.2.1. BASIC EDGE COLOURING ALGORITHM

The aim is to renumber or group the edges in such a way that edges access only once any of the points belonging to the edges in a given group. A simple colouring algorithm that achieves this requirement is given by:

```
 C1. Set a maximum desired vector length  mvecl;
 C2. Initialize new edge number array:  ledge=0;
 C3. Initialize point marking array:  lpoin=0;
 C4. Set new edge counter:  nenew=0;
 C5. Set group counter:  ngrou=0;
 C6. Update group counter:  ngrou=ngrou+1;
 C7. Initialize current vector length counter:  nvecl=0;
 C8. Loop over the edges:
       If the edge  iedge  has not been marked before then:
       If none of the points of this edge have been marked as belonging to this group then:
       - Update new edge counter:  nenew=nenew+1;
       - Store this edge:  lenew(nenew)=iedge;
       - Set  lpoin(ipoin)=ngrou for the points of this edge;
       - Update current vector length counter:  nvecl=nvecl+1;
       Endif
     Endif
     If nvecl.eq.mvecl:  exit edge loop (goto C9);
 C9. Store the group counter:  lgrou(ngrou)=nenew;
C10. If unmarked edges remain (nenew.ne.nedge): goto C6.
```

The speed of this simple colouring algorithm can be improved in a variety of ways. For small vector lengths `mvecl` one may store the smallest unrenumbered edge index `nedg0`. In this way, instead of looping over all the edges in C8, the loop extends over edges `nedg0` to `nedge`. For large vector lengths `mvecl`, one may work with a list of unrenumbered edges. This list is reordered after each pass until all edges have been renumbered. Such a list of unrenumbered edges allows the first `if`-test in C8 to be omitted.

15.2.2. BACKWARD/FORWARD STRATEGY

An immediate improvement to this basic algorithm is obtained by performing two colouring passes over the mesh in opposite directions as shown in Figure 15.13.

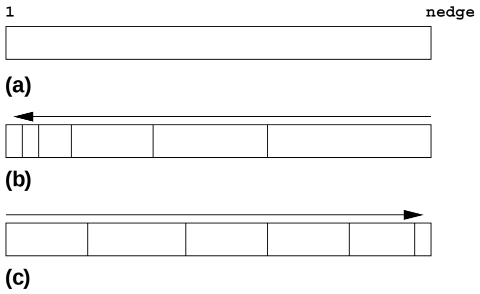


Figure 15.13. Backward/forward strategy: (a) original edge array; (b) backward renumbering pass; (c) forward renumbering pass

The first, ‘backward’ pass is used to determine an appropriate vector length `mvecl=nedge/ngrou` for the second pass. Suppose that the maximum number of edges surrounding a particular point is `mesup`. The minimum number of groups required for independent, vectorizable edge/point operations is then given by `mesup`. At the completion of this first pass, the edges belonging to points with many surrounding edges are stored at the beginning. In the second, ‘forward’ pass over the edges a more balanced grouping of edges is achieved.

15.2.3. COMBINING VECTORIZABILITY WITH DATA LOCALITY

For machines with a small number of vector processors, a good way to minimize cache-misses while looping over vectorizable groups of edges is to choose a relatively small vector length (e.g. `mvecl=256` for 2 processors) and step through the point data as monotonically as possible. This is done by first renumbering the edges according to their minimum point number (see above), and then using the basic colouring scheme or the backward/forward scheme with `mvecl=128`. The data access required for edges grouped in this way is shown in Figure 15.14. Because edge data and point data increase in parallel, cache-misses are avoided whilst allowing for vectorization.

The resulting edge groups will access point information according to the ranges shown schematically in Figure 15.15.

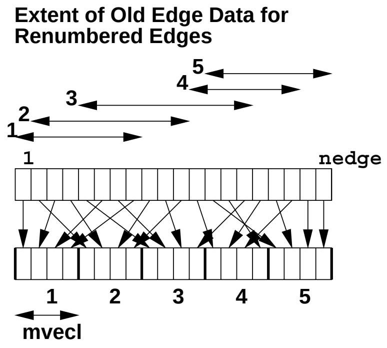


Figure 15.14. Renumbering for vectorizability and data locality

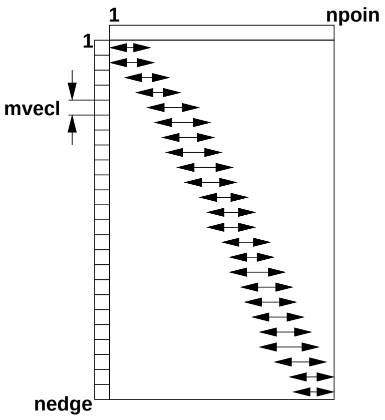


Figure 15.15. Point range accessed by edge groups

15.2.4. SWITCHING ALGORITHM

A typical one-processor register-to-register vector machine will work at its peak efficiency if the vector lengths are multiples of the number of vector registers. For a traditional Cray, this number was 64. An easy way to achieve higher rates of performance without a large change in architecture is through the use of multiple vector pipes. A Cray-C90, for example, has two vector pipes per processor, requiring vector lengths that are multiples of 128 for optimal hardware use. For a NEC-SX4, which has four vector pipes per processor, this number rises to 256, and for the NEC SX8 to 512. The most common way to use multiple processors in these systems is through autotasking, i.e. by splitting the work at the `DO`-loop level. Typically, this is done by simply setting a flag in the compiler. The acceptable vector length for optimal machine utilization is now required to increase again according to the number of processors. For the Cray-C90, with 16 processors, the vector lengths have to be multiples of 2048. The message is clear: for shared- and fixed-memory machines, make the vectors as long as possible while achieving lengths that are multiples of a possibly large machine-dependent number.

Consider a typical unstructured mesh. Suppose that the maximum number of edges surrounding a particular point is `mesup`. The minimum number of groups required for a vectorizable scatter-add pass over the edges is then given by `mesup`. On the other hand, the average number of edges surrounding a point `aesup` will in most cases be lower. Typical numbers are `aesup=6` for linear triangles, `aesup=22` for linear tetrahedra, `aesup=4` for bilinear quads and `aesup=8` for trilinear bricks. For a traditional edge renumberer like the one presented in the previous section, the result will be a grouping or colouring of edges consisting of `aesup` large groups of approximately `nedge/aesup` edges, and some very small groups for the remaining edges. While this way of renumbering is optimal for memory-to-memory machines, better ways are possible for register-to-register machines.

Consider the following 'worst-case scenario' for a triangular mesh of `nedge=391` edges with a grouping of edges according to

`lgrou(1:ngrou)=65,65,65,65,65,65,1`.

For a machine with 64 vector registers this is equivalent to the following grouping:

`lgrou(1:ngrou)=64,64,64,64,64,64,1,1,1,1,1,1,1`,

obtained by setting `mvecl=64` in C1 above, which clearly is suboptimal. The vector registers are loaded up seven times with only one entry. One option is to treat the remaining seven edges in scalar mode. Clearly the optimal way to group edges is

`lgrou(1:ngrou)=64,64,64,64,64,64,7`,

which is possible due to the first grouping. On the other hand, a simple forward or even a backward–forward pass renumbering will in most cases not yield this desired grouping. One possible way of achieving this optimal grouping is the following switching algorithm that is run after a first colouring is obtained (see Figure 15.16).

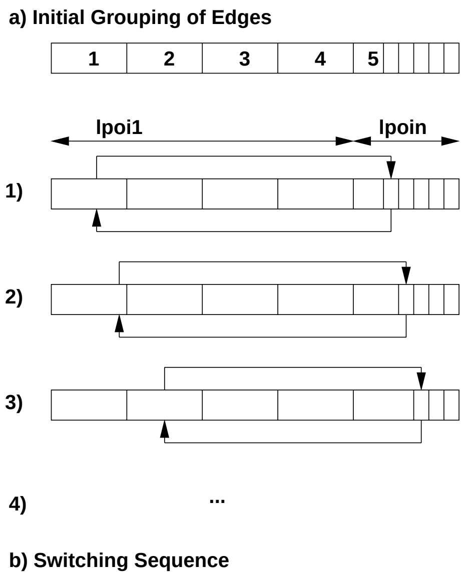


Figure 15.16. Switching algorithm

The idea is to interchange edges at the end of the list with some edge inside one of the vectorizable groups without incurring a multiple point access. Algorithmically, this is accomplished as follows.

```
 S1. Determine the last group of edges with acceptable length; this group is denoted
     by ngro0; the edges in the next group will be nedg0=lgrou(ngro0)+1
     to nedg1=lgrou(ngro0+1), and the remaining edges will be at locations
     nedg1+1 to nedge;
 S2. Transcribe edges nedg1+1 to nedge into an auxiliary storage list;
 S3. Set a maximum desired vector length mvecl;
 S4. Initialize a point array: lpoin=0;
 S5. For all the points belonging to edges nedg0 to nedg1: set lpoin=1;
 S6. Set current group vector length mvecl=nedg1-nedg0+1;
 S7. Set remaining edge counter ierem=nedg1;
 S8. Loop over the large groups of edges igrou=1,ngro0;
 S9. Initialize a second point array: lpoi1=0;
S10. For all the points belonging to the edges in this group: set lpoi1=1;
S11. Loop over the edges iedge in this group:
     If lpoin=0 for all the points touching iedge: then
     Set lpoi1=0 for the points touching iedge;
     If lpoin=0 for all the points touching ierem: then
     - Set lpoin=1 for the points touching iedge;
     - Set lpoi1=1 for the points touching ierem;
     - Interchange edges
     - Update remaining edge counter: ierem=ierem+1
     - Update current vector length counter: nvecl=nvecl+1;
     - If nvecl.eq.mvecl: exit edge loop (Goto S10);
     Else
     - Re-set lpoi1=1 for the points touching iedge;
     Endif
   End of loop over the large groups of edges
S12. Store the group counter: lgrou(ngrou)=nenew;
S13. If unmarked edges remain (ienew.ne.nedge): reset counters and Goto S1.
```

All of the renumberings described work in linear time complexity and are straightforward to code. Thus, they are ideally suited for applications requiring frequent mesh changes, e.g. adaptive h-refinement (Löhner and Baum (1992)) or remeshing (Löhner (1990)) for transient problems. Furthermore, although the ideas were exemplified on edge-based solvers, they carry over to element- or face-based solvers.

15.2.5. REDUCED I/A LOOPS

Suppose the edges are defined in such a way that the first point always has a lower point number than the second point. Furthermore, assume that the first point of each edge increases with stride one as one loops over the edges. In this case, Loop 1 (or the inner loop of Loop 2) may be rewritten as follows.

Loop 1a:

```
      do 1600 iedge=nedg0,nedg1
      ipoi1=kpoi0+iedge
      ipoi2=lnoed(2,iedge)
      redge=geoed(  iedge)*(unkno(ipoi2)-unkno(ipoi1))
      rhspo(ipoi1)=rhspo(ipoi1)+redge
      rhspo(ipoi2)=rhspo(ipoi2)-redge
1600 continue
```

Compared to Loop 1, the number of i/a fetch and store operations has been *halved* while the number of FLOPS remains unchanged. However, unlike stars, superedges and chains, the basic connectivity arrays remain unchanged, implying that a progressive rewrite of codes is possible. In the following, a point and edge renumbering algorithm is described that seeks to maximize the number of loops of this kind that can be obtained for tetrahedral grids.

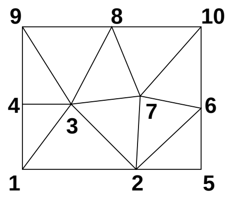
15.2.5.1. Point renumbering

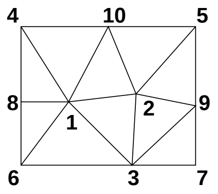
In order to obtain as many uniformly-accessed edge groups as possible, the number of edges attached to a point should decrease uniformly with point number. In this way, the probability of obtaining an available second point to avoid memory contention in each vector group is maximized. The following algorithm achieves such a point renumbering:

```
 Initialization:
From the edge-connectivity array lnoed:
Obtain the points that surround each point;
Store the number of points surrounding each point: lpsup(1:npoin);
Set npnew=0;

 Point Renumbering:
- while(npnew.ne.npoin):
- Obtain the point ipmax with the maximum value of lpsup(ip);
- npnew=npnew+1                          ! update new point counter
- lpnew(npnew)=ipmax                         ! store the new point
- lpsup(ipmax)=   0                             ! update lpsup
- do: for all points jpoin surrounding ipmax:
  - lpsup(jpoin)=max(0,lpsup(jpoin)-1)
- enddo
- endwhile
```

This point renumbering is illustrated in Figures 15.17(a) and (b) for a small 2-D example. The original point numbering, together with the resulting list of edges, is shown in Figure 15.17(a). Renumbering the points according to the maximal number of available neighbours results in the list of points and edges shown in Figure 15.17(b). One can see that the list of edges attached to a point decreases steadily, making it possible to achieve large vector lengths for loops of type 1a (see above).





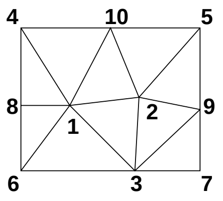
Point	Connections			
1	2	3	4	
2	3	5	6	7
3	4	7	8	9
4	9			
5	6	7		
6	7	10		
7	8	10		
8	9	10		
9				
10				

(a)

Point	Connections					
1	2	3	4	6	8	10
2	3	5	9	10		
3	6	7	9			
4	8	10				
5	9	10				
6	8					
7	9					
8						
9						
10						

(b)

Figure 15.17. (a) Original and (b) maximal connectivity point renumbering



Point	Connections					
1	10	8	6	4	2	3
2	9	10	5	$\underline{3}$		
3	7	$\underline{9}$	7	6		
4	$\underline{8}$	10				
5	10	9				
6	8					
7	9					
8						
9						
10						

Figure 15.18. Reordering into vector groups

15.2.5.2. Edge renumbering

Once the points have been renumbered, the edges are reordered according to point numbers as described above (section 15.1.4). Thereafter, they are grouped into vector groups to avoid memory contention (sections 15.2.1–15.2.3). In order to achieve the maximum (ordered) vector length possible, the highest point number is processed first. In this way, memory contention is delayed as much as possible. The resulting vector groups obtained in this way for the small 2-D example considered above is shown in Figure 15.18.

It is clear that not all of these groups will lead to a uniform stride access of the first point. These loops are still processed as before in Loop 1. The final form for the edge loop then takes the following form.

Loop 2a:

```
     do 1400 ipass=1,npass
     nedg0=edpas(ipass)+1
     nedg1=edpas(ipass+1)
     kpoi0=lnoed(1,nedg0)-nedg0
     idiff=lnoed(1,nedg1)-nedg1-kpoi0
     if(idiff.ne.0) then
cdir$ ivdep
         do 1600 iedge=nedg0,nedg1
         ipoi1=lnoed(1,iedge)
         ipoi2=lnoed(2,iedge)
         redge=geoed(  iedge)*(unkno(ipoi2)-unkno(ipoi1))
         rhspo(ipoi1)=rhspo(ipoi1)+redge
         rhspo(ipoi2)=rhspo(ipoi2)-redge
 1600   continue
       else
cdir$  ivdep
          do 1610 iedge=nedg0,nedg1
          ipoi1=kpoi0+iedge
          4poi2=lnoed(2,iedge)
          redge=geoed(  iedge)*(unkno(ipoi2)-unkno(ipoi1))
          rhspo(ipoi1)=rhspo(ipoi1)+redge
          rhspo(ipoi2)=rhspo(ipoi2)-redge
 1610    continue
       endif
 1400 continue
```

As an example, a F117 configuration is considered, typical of inviscid compressible flows. This case had 619 278 points, 3 509 926 elements and 4 179 771 edges. Figure 15.19 shows the surface mesh. Table 15.10 lists the number of edges processed in reduced i/a mode as a function of the desired vector length chosen. The table contains two values: the first is obtained if one insists on the vector length chosen; the second is obtained if the usual i/a vector groups are examined further, and snippets of sufficiently long (>64) reduced i/a edge groups are extracted from them. Observe that, even with considerable vector lengths, more than 90% of the edges can be processed in reduced i/a mode.

15.2.5.3. Avoidance of cache-misses

Renumbering the points according to their maximum connectivity, as required for the reduced i/a point renumbering described above, can lead to very large jumps in the point index for an edge (or, equivalently, the bandwidth of the resulting matrix).

One can discern that for the structured mesh shown in Figure 15.20 the maximum jump in point index for edges is $nb_{max} = O(N_p/2)$, where N_p is the number of points in the mesh.

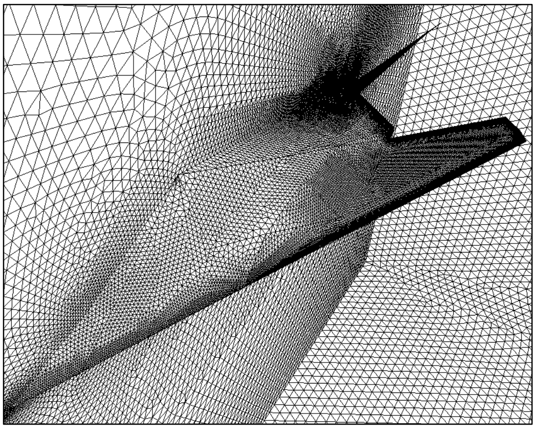


Figure 15.19. F117: surface discretization

Table 15.10. F117 Configuration: `nedge=4,179,771`

`mvecl`	% reduced i/a	`nvecl`	% reduced i/a	`nvecl`
64	97.22	63	97.22	63
128	93.80	127	98.36	121
256	89.43	255	97.87	223
512	86.15	510	97.24	384
1024	82.87	1018	96.77	608
2048	79.30	2026	96.44	855
4096	76.31	4019	96.05	1068
8192	73.25	7856	95.35	1199
16384	67.16	15089	92.93	1371

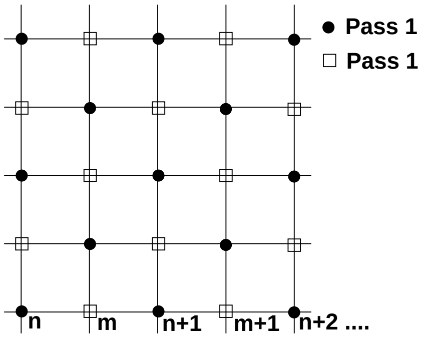


Figure 15.20. Point jumps per edge for a structured grid

In general, the maximum jump will be $nb_{\max} = O((1-2/nc_{\max})N_p)$, where $nc_{\max}$ is the maximum number of neighbours for a point. For tetrahedral grids, the average number of neighbours for a point is approximately $nc = 14$, implying $nb = O(6/7N_p)$. This high jump in point index per edge in turn leads to a large number of cache-misses and consequent loss of performance for RISC-based machines. In order to counter this effect, a two-step procedure is employed. In a first pass, the points are renumbered for optimum cache performance using a bandwidth minimization technique (Cuthill–McKee, wave front, recursive bisection, space-filling curve, bin, coordinate-based, etc.). In a second pass, the points are renumbered for optimal i/a reduction using the algorithm outlined above. However, the algorithm is applied progressively on point groups `npoi0:npoi0+ngrou`, until all points have been covered. The size of the group `ngrou` corresponds approximately to the average bandwidth. In this way, the point renumbering operates on only a few hyperplanes or local zones at a time, avoiding large point jumps per edge and the associated cache-misses.

15.2.6. ALTERNATIVE RHS FORMATION

A number of test cases (Löhner and Galle (2002)) were run on both the CRAY-SV1 and NEC-SX5 using the conventional Loop 2 and Loop 2a. The results were rather disappointing: Loop 2a was slightly more expensive than Loop 2, even for moderate (>256) vector lengths and more than 90% of edges processed in reduced i/a mode. Careful analysis on the NEC-SX5 revealed that the problem was not in the fetches, but rather in the stores. Removing one of the stores almost doubled CPU performance. This observation led to the unconventional formation of the RHS with two vectors.

`Loop 2b`:

```
       do 1400 ipass=1,npass
       nedg0=edpas(ipass)+1
       nedg1=edpas(ipass+1)
cdir$ ivdep                          ! Cray Ignore Vector DEPendencies
         do 1600 iedge=nedg0,nedg1
         ipoi1=lnoed(1,iedge)
         ipoi2=lnoed(2,iedge)
         redge=geoed(  iedge)*(unkno(ipoi2)-unkno(ipoi1))
         rhsp0(ipoi1)=rhsp0(ipoi1)+redge
         rhsp1(ipoi2)=rhsp1(ipoi2)-redge
  1600  continue
  1400 continue
```

Apparently, the compiler (and this seems to be more the norm than the exception) cannot exclude that `ipoi1` and `ipoi2` are identical. Therefore, the fetch of `rhspo(ipoi2)` in Loop 2b has to wait until the store of `rhspo(ipoi1)` has finished. The introduction of the dual RHS enables the compiler to schedule the load of `rhsp1(ipoi2)` earlier and to hide the latency behind other operations. Note that if `rhsp0` and `rhsp1` are physically identical, no additional initialization or summation of the two arrays is required. The same use of dual RHS vectors implemented in Loop 2b is denoted as Loop 2br in what follows. Finally, the `if`-test in Loop 2a may be removed by reordering the edge-groups in such a way that all usual i/a groups are treated first and all reduced i/a thereafter. This loop is denoted as Loop 2bg.

As an example for the kind of speedup that can be achieved with this type of modified, reduced i/a loop, a sphere close to a wall is considered, typical of low-Reynolds-number

Table 15.11. Sphere close to the wall: `nedge=328,634`

`mvecl`	% reduced i/a	`nvecl`	% reduced i/a	`nvecl`
128	89.53	126	94.88	119
256	87.23	251	94.94	218
512	84.50	490	94.69	371
1024	78.58	947	93.10	592
2048	69.85	1748	90.85	797

incompressible flows. Figure 15.21 shows the mesh in a plane cut through the sphere. This case had 49 574 points, 272 434 elements and 328 634 edges. Table 15.11 shows the number of edges processed in reduced i/a mode as a function of the desired vector length chosen.

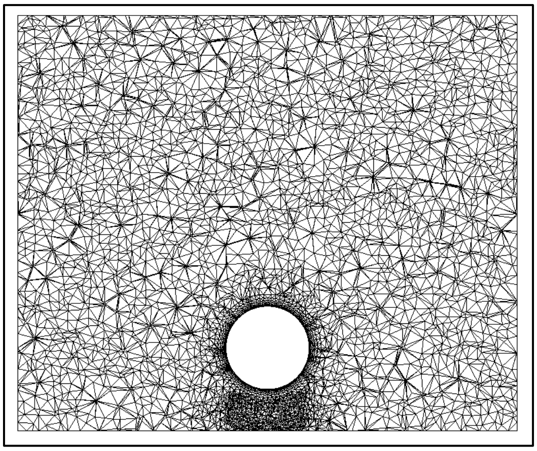


Figure 15.21. Sphere in the wall proximity: mesh in the cut plane

Table 15.12 shows the relative timings recorded for a desired edge group length of 2048 on the SGI Origin2000, Cray-SV1 and NEC-SX5. One can see that gains are achieved in all cases, even though these machines vary in speed by approximately an order of magnitude, and the SGI has an L1 and L2 cache, i.e. no direct memory access. The biggest gains are achieved on the NEC-SX5 (almost 30% speedup).

Table 15.12. Laplacian RHS evaluation (relative timings)

Loop type	O2K	SV1	SX5
Loop 2	1.0000	1.0000	1.0000
Loop 2a	0.9563	1.0077	0.8362
Loop 2b	0.9943	0.8901	0.7554
Loop 2br	0.9484	0.8416	0.7331
Loop 2bg	—	—	0.7073

Table 15.13. Speedups obtainable (Amdahl's Law)

R_p/R_s	50%	90%	99%	99.9%
10	1.81	5.26	9.17	9.91
100	1.98	9.90	50.25	90.99
1000	2.00	9.91	90.99	500.25

15.3. Parallel machines: general considerations

With the advent of massively parallel machines, i.e. machines in excess of 500 nodes, the exploitation of parallelism in solvers has become a major focus of attention. According to Amdahl's Law, the speedup s obtained by parallelizing a portion α of all operations required is given by

$$s = \frac{1}{\alpha \cdot (R_s/R_p) + (1-\alpha)}, \tag{15.1}$$

where R_s and R_p denote the scalar and parallel processing rates (speeds), respectively. Table 15.13 shows the speedups obtained for different percentages of parallelization and different numbers of processors.

Note that even on a traditional shared-memory, multi-processor vector machine, such as the Cray-T90 with 16 processors, the maximum achievable speedup between scalar code and parallel vector code is a staggering $R_p/R_s = 240$. What is important to note is that as one migrates to higher numbers of processors, only the embarrassingly parallel codes will survive. Most of the applications ported successfully to parallel machines to date have followed the single program multiple data (SPMD) paradigm. For grid-based solvers, a spatial sub-domain was stored and updated in each processor. For particle solvers, groups of particles were stored and updated in each processor. For obvious reasons, load balancing (Williams (1990), Simon (1991), Mehrota *et al.* (1992), Vidwans *et al.* (1993)) has been a major focus of activity.

Despite the striking successes reported to date, only the simplest of all solvers, explicit timestepping or implicit iterative schemes, perhaps with multigrid added on, have been ported without major changes and/or problems to massively parallel machines with distributed memory. Many code options that are essential for realistic simulations are not easy to parallelize on this type of machine. Among these, we mention local remeshing (Löhner (1990)), repeated h-refinement, such as that required for transient problems (Löhner and Baum (1992)), contact detection and force evaluation (Haug *et al.* (1991)), some preconditioners (Martin and Löhner (1992), Ramamurti and Löhner (1993)), applications where particles, flow and chemistry interact, and other applications with rapidly varying load imbalances. Even if 99% of all operations required by these codes can be parallelized, the maximum achievable gain will be restricted to 1:100. If we accept as a fact that for most large-scale codes we may not be able to parallelize more than 99% of all operations, the shared-memory paradigm, discarded for a while as non-scalable, may make a comeback. It is far easier to parallelize some of the more complex algorithms, as well as cases with large load imbalance, on a shared-memory machine. Moreover, it is within present technological reach to achieve a 100-processor, shared-memory machine.

15.4. Shared-memory parallel machines

The alternative of having less expensive RISC chips linked via shared memory is currently being explored by a number of vendors. One example of such a machine is the SGI Altix, which at the time of writing allows up to 256 processors to work in shared-memory mode on a problem. In order to obtain proper performance from such a machine, the codes must be written so as to avoid:

(a) cache-misses (in order to perform well on each processor);

(b) cache overwrite (in order to perform well in parallel); and

(c) memory contention (in order to allow pipelining).

Thus, although in principle a good compromise, shared-memory, RISC-based parallel machines actually require a fair degree of knowledge and reprogramming for codes to run optimally.

The reduction of cache-misses and the avoidance of memory contention have already been discussed before. Let us concentrate on the avoidance of cache overwrite. If one desires to port the element loop described above to a parallel, shared-memory machine, the easiest way to proceed is to let the auto-parallelizing compiler simply split the inner loop across processors. It would then appear that increasing the vector length to a sufficiently large value would offer a satisfactory solution. However, this is not advisable for the following reasons.

(a) Every time a parallel `do-loop` is launched, a start-up time penalty, equivalent to several hundred FLOPs is incurred. This implies that if scalability to even 16 processors is to be achieved, the vector loop lengths would have to be 16×1000. Typical tetrahedral grids yield approximately 22 maximum vector length groups, indicating that one would need at least $22 \times 16 \times 1000 = 352\,000$ edges to run efficiently.

(b) Because the range of points in each group increases at least linearly with vector length, so do cache misses. This implies that, even though one may gain parallelism, the individual processor performance would degrade. The end result is a very limited, non-scalable gain in performance.

(c) Because the points in a split group access a large portion of the edge array, different processors may be accessing the same cache-line. When a 'dirty cache-line' overwrite occurs, all processors must update this line, leading to a large increase of interprocessor communication, severe performance degradation and non-scalability. Experiments on an 8-processor SGI Power Challenge showed a maximum speedup of only 1:2.5 when using this option. This limited speedup was attributed, to a large extent, to cache-line overwrites.

In view of these consequences, additional renumbering strategies have to be implemented. In the following, two edge group agglomeration techniques that minimize cache-misses, allow for pipelining on each processor and avoid cache overwrite across processors are discussed. Both techniques operate on the premise that the points accessed within each parallel inner edge loop (`1600 loop`) do not overlap.

Before going on, we define `edmin(1:npass)` and `edmax(1:npass)` to be the minimum and maximum points accessed within each group, respectively, `nproc` the number of processors, and `[edmin(ipass),edmax(ipass)]` the point range of each group `ipass`.

15.4.1. LOCAL AGGLOMERATION

The first way of achieving pipelining and parallelization is by processing, in parallel, `nproc` independent vector groups whose individual point range does not overlap. The idea is to renumber the edges in such a way that `nproc` groups are joined together where, for each one of these groups, the point ranges do not overlap (see Figure 15.22).

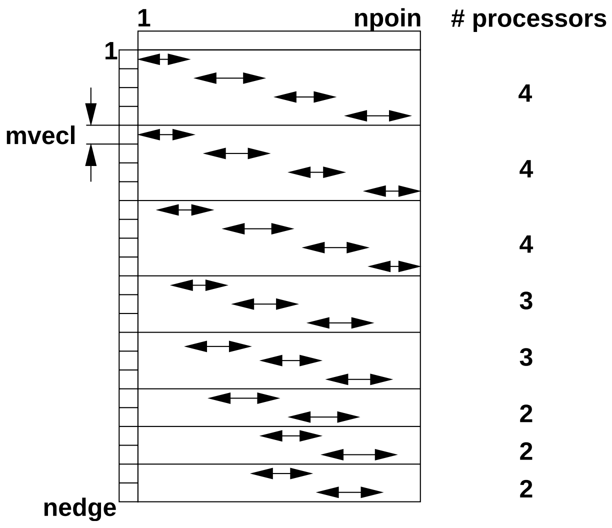


Figure 15.22. Local agglomeration

As each one of the sub-groups has the same number of edges, the load is balanced across the processors. The actual loop is given by the following.

```
Loop 3:
     do 1400 ipass=1,npass
     nedg0=edpas(ipass)+1
     nedg1=edpas(ipass+1)
                                                   ! Parallelization directive
c$omp parallel do private(iedge,ipoi1,ipoi2,redge)
c$dir ivdep                                        ! Pipelining directive
        do 1600 iedge=nedg0,nedg1
        ipoi1=lnoed(1,iedge)
        ipoi2=lnoed(2,iedge)
        redge=geoed(  iedge)*(unkno(ipoi2)-unkno(ipoi1))
        rhspo(ipoi1)=rhspo(ipoi1)+redge
        rhspo(ipoi2)=rhspo(ipoi2)-redge
1600   continue
1400  continue
```

Note that the number of edges in each pass, i.e. the difference `nedg1-nedg0+1`, is now `nproc` times as large as in the original Loop 2. As one can see, this type of renumbering entails no code modifications, making it very attractive for large production codes. However, a start-up cost is incurred whenever a loop across processors is launched. This would indicate that long vector lengths should be favoured. However, cache-misses increase with vector length, so that this strategy only yields a limited speedup. This leads us to the second renumbering strategy.

15.4.2. GLOBAL AGGLOMERATION

A second way of achieving pipelining and parallelization is by processing all the individual vector groups in parallel at a higher level (see Figure 15.23).

In this way, short vector lengths can be kept and low start-up costs are achieved. As before, the point range between macro-groups must not overlap. This renumbering of edges is similar to domain splitting (Flower *et al.* (1990), Williams (1990), Venkatakrishnan *et al.* (1991), Vidwans *et al.* (1993), Löhner and Ramamurti (1995)), but does not require an explicit message passing or actual identification of domains. Rather, all the operations are kept at the (algebraic) array level. The number of sub-groups, as well as the total number of edges to be processed in each macro-group, is not the same. However, the imbalance is small, and does not affect performance significantly. The actual loop is given by the following.

Loop 4:

```
   do 1000 imacg=1,npasg,nproc
   imac0=      imacg
   imac1=min(npasg,imac0+nproc-1)
c                                                ! Parallelization directive
c$omp parallel do private(ipasg,ipass,npas0,
c$&                          npas1,iedge,nedg0,nedg1,
c$&                          ipoi1,ipoi2,redge)
             do 1200 ipasg=imac0,imac1
             npas0=edpag(ipasg)+1
             npas1=edpag(ipasg+1)
             do 1400 ipass=npas0,npas1
             nedg0=edpas(ipass)+1
             nedg1=edpas(ipass+1)
c$dir ivdep                                       ! Pipelining directive
             do 1600 iedge=nedg0,nedg1
             ipoi1=lnoed(1,iedge)
             ipoi2=lnoed(2,iedge)
             redge=geoed(  iedge)*(unkno(ipoi2)-unkno(ipoi1))
             rhspo(ipoi1)=rhspo(ipoi1)+redge
             rhspo(ipoi2)=rhspo(ipoi2)-redge
 1600    continue
 1400   continue
 1200  continue
 1000 continue
```

As one can see, this type of renumbering entails two outer loops, implying that a certain amount of code rewrite is required. On the other hand, the original code can easily by retrieved

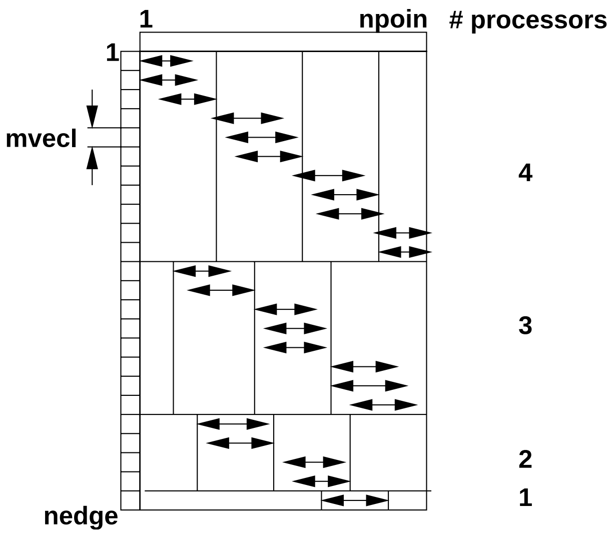


Figure 15.23. Global agglomeration

by setting `edpag(1)=0`, `edpag(2)=npass`, `npasg=1` for conventional uni-processor machines.

If the length of the cache-line is known, one may relax the restriction of non-overlapping point ranges to non-overlapping cache-line ranges. This allows for more flexibility, and leads to almost perfect load balance. A simple edge renumbering scheme that has been found to be effective is the following multipass algorithm:

S1. *Pass 1*: Agglomerate in groups of edges with point range `npoin/nproc`, setting the maximum number of groups in each macro-group `naggl` to the number of groups obtained for the range `1:npoin/nproc`;

S2. *Passes 2ff*: For the remaining groups: determine the point range;

Estimate the range of the first next macro-group from the point range and the number of processors;

Agglomerate the groups in this range to obtain the first next macro-group, and determine the number of groups for each macro-group in this pass `naggl`;

March though the remaining groups of edges, attempting to obtain macro-groups of length `naggl`.

Although not optimal, this simple strategy yields balanced macro-groups, as can be seen from the examples below. Obviously, other renumbering or load balancing algorithms are possible, as evidenced by a large body of literature (see, e.g., Flower *et al.* (1990), Williams (1990), Venkatakrishnan *et al.* (1991), Vidwans *et al.* (1993), Löhner and Ramamurti (1995)).

15.4.3. IMPLEMENTATIONAL ISSUES

For large-scale codes, having to re-write and test several hundred subroutines can be an onerous burden. To make matters worse, present compilers force the user to declare explicitly the local and shared variables. This is easily done for simple loops such as the one described above, but can become involved for the complex loops with many scalar temporaries that characterize advanced CFD schemes written for optimal cache reuse. We have found that in some cases, compilers may refuse to parallelize code that has all variables declared properly. A technique that has always worked, and that reduces the amount of variables to be declared, is to write sub-subroutines. For Loop 4, this translates into the following.

`Master loop 4`:

```
     do 1000 imacg=1,npasg,nproc
     imac0=       imacg
     imac1=min(npasg,imac0+nproc-1)
c                                                    ! Parallelization directive
c$omp parallel do private(ipasg)
      do 1200 ipasg=imac0,imac1
      call loop2p(ipasg)
 1200  continue
 1000 continue
```

`Loop2p`: becomes a subroutine of the form:

```
       subroutine loop2p(ipasg)

       npas0=edpag(ipasg)+1
       npas1=edpag(ipasg+1)

       do 1400 ipass=npas0,npas1
       nedg0=edpas(ipass)+1
       nedg1=edpas(ipass+1)
c$dir ivdep ! Pipelining directive
       do 1600 iedge=nedg0,nedg1
       ipoi1=lnoed(1,iedge)
       ipoi2=lnoed(2,iedge)
       redge=geoed(  iedge)*(unkno(ipoi2)-unkno(ipoi1))
       rhspo(ipoi1)=rhspo(ipoi1)+redge
       rhspo(ipoi2)=rhspo(ipoi2)-redge
1600  continue
1400 continue
```

In order to demonstrate the performance of the agglomeration technique described above, a supersonic inlet problem is considered. The domain, as well as the solution after 800 timesteps are shown in Figures 15.24. The mesh had approximately 540 000 tetrahedra, 106 000 points, 30 000 boundary points and 680 000 edges. After renumbering, the maximum and average bandwidths were 1057 and 468. The vector loop length was set to 16.

Table 15.14. Inlet problem on two processors

`ipasg`	`min(loopl)`	`max(loopl)`
1–2	237 776	237 776
3–4	71 344	71 344
5–6	21 392	21 392
7–8	6416	6416
9–10	1770	1936
11–12	112	1024
13–14	0	576

The minimum and maximum numbers of edges processed for each pass over the processors are given in Tables 15.14–15.17. The corresponding percentage of edges processed by the first processor, as well as the theoretical loss of performance due to imbalance are shown in Table 15.18. Table 15.19 summarizes the speedups obtained for a run of 100 timesteps, including i/o, renumbering, etc., for an SGI Origin 2000 machine. Although this machine is not a true shared-memory machine, it exhibits very fast inter-processor transfer rates, making it possible to achieve reasonable speedups in shared-memory mode. The same run was repeated using domain decomposition and message passing under PVM (parallel virtual machine message passing protocol). Observe that although the PVM-run achieves better speedup, the shared-memory run is still competitive. For large-scale industrial applications of shared-memory parallel machines in conjunction with advanced compressible flow solvers, see Löhner *et al.* (1999) and Rice *et al.* (2000).

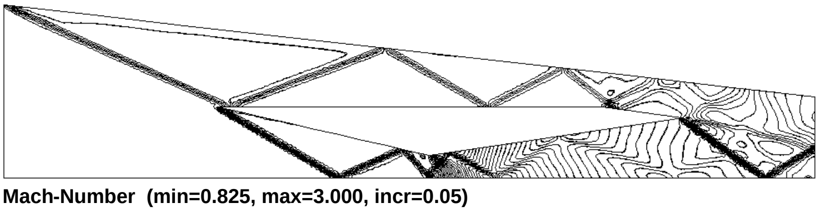


Figure 15.24. Supersonic inlet

15.5. SIMD machines

SIMD machines, as exemplified by the Thinking Machines CM-series, operate very efficiently on nearest-neighbour transfer of operations. On the other hand, general data exchange operations, as required for unstructured grids, take a very large amount of time. A simple gather takes the equivalent of 20–60 FLOPS. Several routers or pre-compilers have been devised to alleviate this problem. At the same time, renumbering strategies have been explored. Both approaches combined lead to a significant decrease in indirect addressing overhead. On the other hand, they are useless for more general applications where the mesh topology changes every few timesteps (remeshing, h-refinement, etc.). Therefore, only a few

Table 15.15. Inlet problem on four processors

`ipasg`	`min(loopl)`	`max(loopl)`
1–4	118 880	118 880
5–8	35 664	35 664
9–12	10 704	10 704
13–16	3216	3216
17–20	266	1024
21–24	800	1024
25–28	0	256

Table 15.16. Inlet problem on six processors

`ipasg`	`min(loopl)`	`max(loopl)`
1–6	79 248	79 248
7–12	23 776	23 776
13–18	7136	7136
19–24	0	2144
25–30	1024	1024
31–36	0	1024
37–42	0	336

Table 15.17. Inlet problem on eight processors

`ipasg`	`min(loopl)`	`max(loopl)`
1–8	59 440	59 440
9–16	17 824	17 824
17–24	5360	5360
25–32	650	1600
33–40	0	1024
41–48	0	1024
49–56	0	288

Table 15.18. Inlet: actual versus optimal edge allocation

`nproc`	Actual %	Optimal %	Loss %
2	50.122	50.00	0.24
4	25.140	25.00	0.56
6	16.884	16.67	1.01
8	12.743	12.50	1.94

Table 15.19. Speedups for inlet problem

`nproc`	Speedup (shared)	Speedup (PVM)
2	1.81	1.83
4	3.18	3.50
6	4.31	5.10
8	5.28	—

steady-state or fixed mesh transient applications have been successful on this type of machine (see, e.g., Jespersen and Levit (1989), Long *et al.* (1989), Oran *et al.* (1990)). SIMD machines may be compared to memory-to-memory vector machines: they inherently lack generality, which may lead to their eventual demise.

15.6. MIMD machines

MIMD machines, as exemplified by the Intel, NCube, Parsytech, etc. hypercubes, the Intel Paragon mesh and clusters of compute nodes (IBM-SP) or simply PCs (so-called Beowulf clusters), at present consist of fairly powerful processors that are linked by message passing and synchronization software. In the future, each of these processors will be a vector processor. This implies that most of the algorithmic considerations discussed for scalar and vector machines will carry over to them. If one considers hundreds or even thousands of processors, it becomes clear that, in order to avoid memory access bottlenecks, both the CPU power and the memory must be spread among the processors. Architectures of this type are denoted as distributed-memory parallel machines. In the following, we will base all considerations on the assumption of such an architecture. The main issues when trying to code optimally for this type of architecture are as follows.

(a) *Minimization of idle time*. In the worst-case scenario, all but one processor wait for the last processor to finish a certain task. This implies that one has to strive for the same amount of work and communication in each processor.

(b) *Minimization of interprocessor information flow*. With processor performance advancing rapidly in comparison to interprocessor transfer speeds, the amount of information required between processors has to be minimized. In general, this will lead to a minimization problem for the area-to-volume ratio of the domain treated by the processors.

(c) *Extra layers/additional work trade-offs*. The amount of required information transfer between processors can be staged (usual code) or more information can be gathered ahead of the timestep. In the latter case, no further information transfer is required within a timestep. On the other hand, more layers of information surrounding each domain are required. Both approaches are possible, and the performance of each may depend more on the individual hardware of each machine than anything else.

15.6.1. GENERAL CONSIDERATIONS

The effective use of any parallel machine requires the following general steps for the solution of the problem at hand:

P1. break up the problem to be solved into pieces;

P2. hand each processor a piece of the problem;

P3. if required, provide for interprocessor transfer of information;

P4. assemble the results.

The processor hierarchy and scheduling may also vary. Some of the possible choices are:

- all processors working at the same level (used for grid smoothing and most flow solvers);
- a pyramid of masters that hand out and/or perform tasks;
- a one master – many slaves doing different operations hierarchy (used for unstructured grid generation);
- a one master – many slaves doing the same operation hierarchy (the SIMD paradigm).

Porting a typical flow code to a MIMD requires the following pieces of software:

- parallel input modules;
- parallel domain subdivision modules for load balancing;
- node-versions of the flow code for parallel execution;
- interdomain info-transfer modules;
- parallel adaptive grid regeneration modules;
- parallel H-refinement modules;
- parallel output modules.

The message (with all its associated consequences) is clear: do everything in parallel, or do not even start.

15.6.2. LOAD BALANCING AND DOMAIN SPLITTING

For field solvers based on grids, parallel execution with distributed memory machines is best achieved by domain splitting. The elements are grouped together into sub-domains, and are allocated to a processor. Each sub-domain can then advance the solution independently, enabling parallel execution. At the boundaries of these sub-domains, information needs to be transferred as the computation proceeds. Because of this association between processors and sub-domains, these terms will be used interchangeably within the same context.

The computational work required by field solvers is proportional to the number of elements. This proportionality is linear for explicit time-marching or iterative schemes, of the $N \log(N)$ type for multigrid solvers, and of the $N^p,\ p > 1$ type for implicit time-marching or direct solvers. The factor p depends on the bandwidth of the domain, which in turn not only depends on the number of elements, but also the shape of the domain, the ratio of the number of boundary points to the domain points, the local numbering of the points and elements, etc. This implies that the optimal load balance is achieved if each processor is allocated the same number of elements, and all processors have sub-domains with approximately the same shape. Besides the CPU time required in each processor to advance the solution to the next level (an iteration, a timestep), the communication overhead between processors has to be taken into account. This overhead is directly proportional to the amount of information that needs to be transferred between processors. For field solvers based on grids, this amount of information is proportional to the number of sub-domain boundary points. Therefore, an optimal load and communication balance is achieved by allocating to each processor the same number of elements while minimizing the number of boundary points in each sub-domain.

In many situations, the computational work required in an element or sub-domain of elements may change by orders of magnitude during a run. For grids that are continuously refined/derefined every 5 to 10 timesteps (e.g. strongly unsteady flows with travelling shock waves (Löhner and Baum (1992)), the number of elements in a sub-domain may increase or decrease rapidly. With only two levels of h-refinement, the number of elements increases by a factor of 64 in three dimensions. For flows that are chemically reacting, the CPU requirements can vary by factors in excess of 100 depending on whether the element is in the cold, unreacted region, the flame, or the hot, reacted region (Patnaik *et al.* (1991)). Both of these cases illustrate the need for fast, dynamic load balancing algorithms.

A number of load balancing algorithms have been devised in recent years. They can be grouped into three families: simulated annealing (Dahl (1990), Flower *et al.* (1990), Williams (1990)), recursive bisection (Williams (1990), von Hanxleden and Ramamurti (1991), Simon (1991), Venkatakrishnan *et al.* (1991), Vidwans *et al.* (1993)) and diffusion (Cybenko (1989), Pruhs *et al.* (1992), Löhner and Ramamurti (1995)). In the following, a brief description of each of these algorithms is given.

15.6.2.1. Simulated annealing

This technique, borrowed from the simulation of atoms in a body in thermal equilibrium at finite temperatures, tries to achieve optimal load balancing by randomly changing discrete pieces of work (e.g. elements for a mesh) among processors. In principle, only the changes that lead to a better work allocation should be accepted. However, this may force the algorithm to stop in local minima, never leading to a global minimum. Therefore, changes that lead to a work allocation that is worse than the present one may also be accepted. The acceptance is based on the comparison of a random number in the interval [0, 1] with a probability function that diminishes as the algorithm proceeds (Dahl (1990), Flower *et al.* (1990), Williams (1990)). The improvement of load balancing is usually measured by a cost function that takes into account both the work for each processor, as well as the communication between processors. The main advantage of this class of technique is its simplicity. However, because it is based largely on randomness and not on ratio, many attempts have to be made to achieve a load distribution that is close to optimal. This makes it very expensive (Williams (1990)), particularly for applications that require many load redistributions during a run.

15.6.2.2. Recursive bisection

Recursive bisection techniques achieve load balancing by recursively subdividing the work among processors. For domain solvers, the most natural way to subdivide the work is by subdividing the mesh. Several ways of bisecting a mesh have been explored (Williams (1990), von Hanxleden and Scott (1991), Simon (1991), Venkatakrishnan *et al.* (1991)). The most popular of these are the following.

- *Orthogonal recursive bisection (ORB)*: the elements, cells or points in a sub-domain are split according to their spatial coordinates. In order to preserve good surface-to-volume ratios (low communication overhead), a cyclic change of coordinates is employed. Figure 15.25 illustrates the typical subdivision sequence for a mesh given eight processors.
- *Cuthill–McKee recursive bisection (CRB)*: the elements, cells or points in a sub-domain are first renumbered using the Cuthill–McKee (1969) bandwidth minimization algorithm. This leads to a new list of elements, cells, edges or points. The sub-domain is bisected by simply allocating the first and second half of the relevant list (elements, cells, edges or points) to new sub-domains.
- *Moment recursive bisection (MRB)*: a way to improve the surface-to-volume ratios for the sub-domains as compared to ORB is to compute a moments of inertia matrix

  $$\Theta^{ij} = \sum_{k=1,\ \text{points}} x_k^i x_k^j ,$$

  and then bisect the domain further by halving along the principal direction of Θ.
- *Eigenvalue recursive bisection (ERB)*: in this case the properties of the eigenvalues and eigenvectors of the Laplacian of a graph are exploited to produce a bisection that minimizes the surface-to-volume ratio of the sub-domains. This recursive technique requires the solution of eigenvalue problems and can be quite costly, making it doubtful whether it will be useful for problems that require many dynamic load balancing steps during a run.

The advantages of recursive bisection methods may be summarized as follows.

- They are mathematically elegant, leading to a subdivision of the work for 2^n processors in n steps (10 steps for 1024 processors).
- For domain solvers whose cost is linear in the number of elements, cells, edges or points, recursive bisection methods will lead to optimal load balancing. This would apply to explicit solvers, or iterative solvers with preconditioners whose work requirements grow linearly with the number of unknowns (e.g. conjugate gradient with diagonal preconditioning).

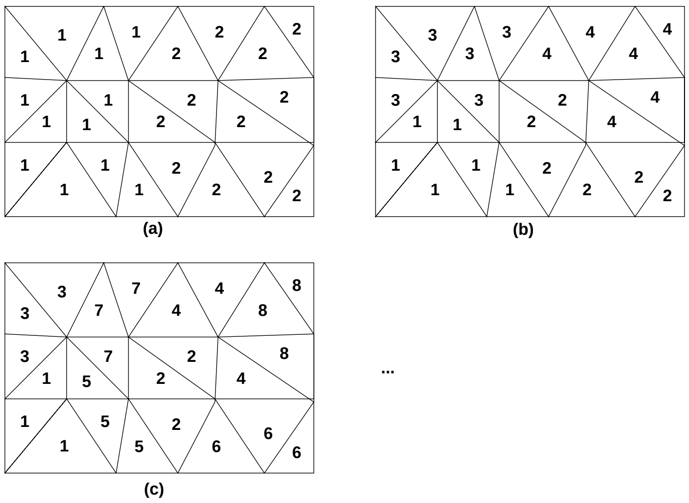


Figure 15.25. Orthogonal recursive bisection: (a) subdivision in x; (b) subdivision in y; (c) subdivision in x

- For domain solvers whose cost is composed from a linear portion in the number of cells, edges or points, and a nonlinear portion that is proportional to the surface-to-volume ratio (e.g. bandwidth), recursive bisection methods will lead to a fairly good load balancing. This is because the bandwidth in each sub-domain will be approximately the same.
- Locally varying workloads in each element, cell, edge or point (e.g. chemically reacting flows) can be accounted for in the OCB, MRB and ERB by weighting the coordinates of the elements, cells, edges or points by their respective work units. For CRB, a possible way to treat varying workloads is to compute the expected work in each sub-domain, halving the domain as it is renumbered wherever half of the work is exceeded.

The disadvantages of recursive bisection methods may be summarized as follows.

- After the new load allocation has been achieved, a global reordering of data is required. This leads to substantial communication overheads. The alternative, to keep the data local and access it via indirect addressing globally, would lead to massive communication overheads, slowing down execution speeds.
- The amount of work required to achieve a new load balance is fixed, i.e. it does not depend on the degree of imbalance between processors. Thus, even though only one processor may suddenly encounter an increase in work, all processors get involved in data transfer.
- Recursive bisection methods, in their basic form, require the number of processors n_p to be of the form $n_p = 2^n$. This may prove difficult to achieve in many parallel processing environments, particularly if networks of workstations or PCs are considered. One could, in principle, devise subdivisions of the form $n_p = n_1 \cdot n_2 \cdot n_3 \cdot \cdots$, but even this more elaborate strategy will not always work satisfactorily (e.g. a prime number of processors).

15.6.2.3. Diffusion methods

The idea here is to achieve load balancing by exchanging workloads (e.g. elements) among processors. Those processors that have too much work to do give more work to their neighbours. The analogy to heat conduction – hence the name diffusion – is obtained by associating work with temperatures. Diffusion methods have so far been treated mainly in the more theoretical computer science literature (Cybenko (1989), Pruhs *et al.* (1992)). The application to practical problems was demonstrated recently by Löhner *et al.* (1995). The main advantage of diffusion methods is the ability to measure exactly, and hence balance out as best as possible, the workload encountered. Compared to recursive bisection, the price to be paid is that several small nearest-neighbour transfers are traded for one large, possibly long distance, data exchange. In this respect, diffusion methods resemble the simulated annealing approach. However, the exchange of elements between sub-domains is steered by a rational improvement strategy as opposed to random trial and error. Löhner *et al.* (1995) show that algorithms of this kind can be devised that converge to good load balance in only a few passes over the mesh, making the diffusion methods highly competitive. A disadvantage of diffusion methods is the possible loss of control over the shape of the boundaries as elements are exchanged during load balancing.

15.6.2.4. Greedy/advancing front

A simple algorithm that attempts to obtain a subdivision which minimizes the surface-to-volume ratio can be constructed using the following 'greedy', 'advancing front' or 'wave front'-type scheme (Venkatakrishnan *et al.* (1991), Löhner and Ramamurti (1995)):

Assume given:

- the elements that surround each point;
- an initial starting element that is on the boundary;
- a real number `reffe` in each element associated with the effort to be spent in it;
- a real number `reffd` that denotes the desired average cumulative effort in each sub-domain.

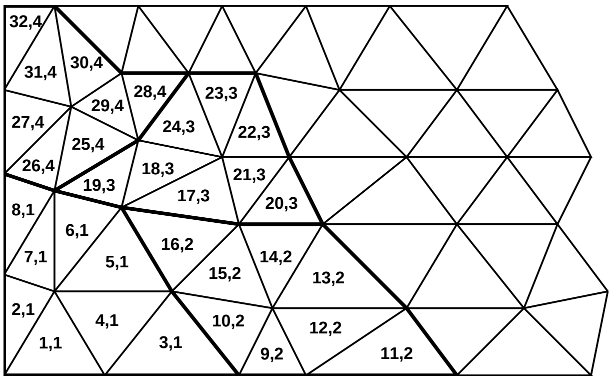


Figure 15.26. Advancing front domain decomposition

```
Then:
S1.  Initialize point and element arrays;
S2.  Initialize domain counter  reffd ;
S3.  Start next sub-domain:
     Update domain number;
     Initialize sub-domain effort counter;
S4. Select the next point to be surrounded from the order of creation list;
S5. Mark this point as totally surrounded;
S6. For each of the elements surrounding this point:
     if:  the element has not been marked as belonging to a domain before:
                - Mark the element as belonging to the present domain;
                - Update sub-domain effort counter;
                - For the nodes of this element:
                  if:  the point is not yet totally surrounded and has not
                  yet been incorporated into the order of creation list:
                      Add the point to the order of creation list;
                endif
endif
S7.  if: the sub-domain effort counter exceeds  reffd:
     Compress point creation list, eliminating all totally surrounded points;
                                                                  Goto S3.
else
                                                                  Goto S4.
endif
```

Figure 15.26 shows how this algorithm works on a simple 2-D domain, where we assumed `reffe=1.0` and `reffd=8.0`.

This simple splitting algorithm can produce isolated ‘islands’ of elements, or disjoined domains. The amount of interprocessor communication may be improved by using a diffusion type algorithm, see, e.g., Löhner *et al.* (1993).

15.6.3. PARALLEL FLOW SOLVERS

Consider first explicit flow solvers, or implicit flow solvers that use iterative techniques. The main operations are the building of the RHS, which entails loops over elements or edges, max/min comparisons among neighbours and the assembly of element or edge-RHS at points.

In order to be able to change the unknowns at sub-domain interfaces, an exchange of information between processors has to be allowed. Two ways of accomplishing this transfer are possible (see Figure 15.27).

(a) *RHS information*. Given by the following set of operations:

- assemble all RHS contributions in each processor;
- exchange RHS contributions between processors, adding;
- update the unknowns at the interfaces.

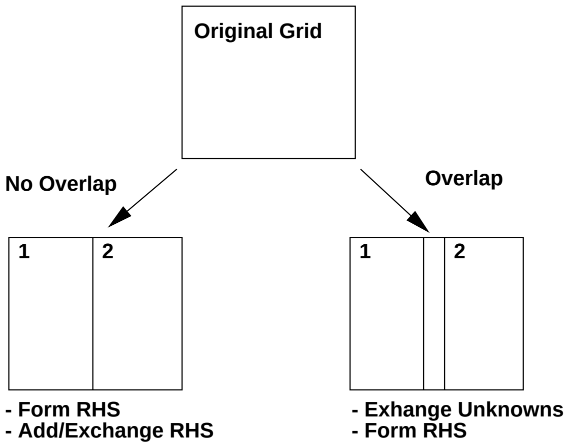


Figure 15.27. RHS assembly for parallel flow solvers

A similar sequence is required for min/max comparisons among nearest neighbours (e.g. for limiting purposes). The advantage of this approach is that the total amount of information (elements, points) required for the parallel version is the same as for the serial version. However, the original serial code has to be modified extensively. These low-level modifications are also reflected in a relatively large number of transfer operations among processors if limiting or other involved nearest neighbour operations are required when building a RHS.

(b) *Unknown information*. If an additional layer of elements is added to each sub-domain, the build-up of a RHS can be accomplished as follows:

- update each sub-domain as for the serial, one-domain case;
- exchange the updated coordinate information between processors.

In this approach, the total amount of information (elements, points) required for the parallel version is larger than that required for the serial version. However, the extra amount of information is not large (one layer of elements). On the other hand, the code employed in each sub-domain is exactly the same as for the serial case. This advantage outweighs all possible disadvantages, and has made this second possibility the method of choice for most codes used in practice.

The second method requires the addition of layers of elements at the boundaries of the sub-domains. The following algorithm accomplishes this task:

Assume given:

- the elements that surround each point;
- the domain number each element belongs to.

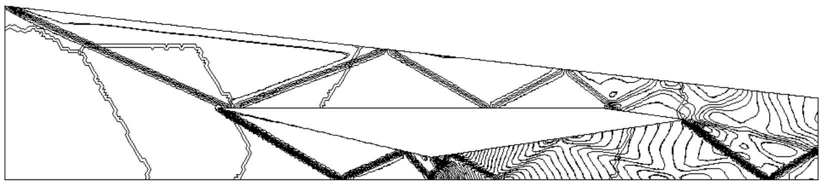
```
Then:
L1. Initialize pointer lists for elements, points and receive lists;
L2. For each point ipoin:
    Get the smallest domain number idmin of the elements that surround it; store this
    number in lpmin(ipoin);
    For each element that surrounds this point:
    If the domain number of this element is larger than idmin:
              - Add this element to domain idmin;
L3. For the points of each sub-domain idomn:
    If  lpmin(ipoin).ne.idomn:
    add this information to the receive list for this sub-domain;
    Endif
L4. Order the receive list of each sub-domain according to sub-domains;
L5. Given the receive lists, build the send list for each sub-domain.
    Given the send and receive lists, the information transfer required for the parallel
    explicit flow solver is accomplished as follows:
    - Send the updated unknowns of all nodes stored in the send list;
    - Receive the updated unknowns of all nodes stored in the receive list;
    - Overwrite the unknowns for these received points.
```

In order to demonstrate the use of explicit flow solvers on MIMD machines, we consider the same supersonic inlet problem as described above for shared-memory parallel machines (see Figure 15.24). The solution obtained on a 6-processor MIMD machine after 800 timesteps is shown in Figure 15.28(a). The boundaries of the different domains can be clearly distinguished. Figure 15.28(b) summarizes the speedups obtained for a variety of platforms using MPI as the message passing library, as well as the shared memory option. Observe that an almost linear speedup is obtained. For large-scale industrial applications of domain decomposition in conjunction with advanced compressible flow solvers, see Mavriplis and Pirzadeh (1999).

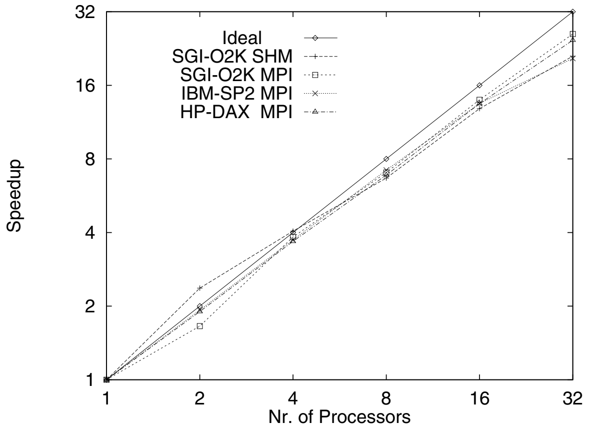
15.7. The effect of Moore's law on parallel computing

One of the most remarkable constants in a rapidly changing world has been the rate of growth for the number of transistors that are packaged onto a square inch. This rate, commonly known as Moore's Law, is approximately a factor of two every 18 months, which translates into a factor of 10 every 5 years (Moore (1965, 1999)). As one can see from Figure 15.29 this rate, which governs the increase in computing speed and memory, has held constant for more than three decades, and there is no end in sight for the foreseeable future (Moore (2003)).

One may argue that the raw number of transistors does not translate into CPU performance. However, more transistors translate into more registers and more cache, both important elements to achieve higher throughput. At the same time, clock rates have increased, and pre-fetching and branch prediction have improved. Compiler development has also not stood still. Moreover, programmers have become conscious of the added cost of memory access, cache misses and dirty cache lines, employing the techniques described above to minimize their impact. The net effect, reflected in all current projections, is that CPU performance is going to continue advancing at a rate comparable to Moore's Law.



(a)



(b)

Figure 15.28. Supersonic inlet: (a) MIMD results; (b) speedup for different machines

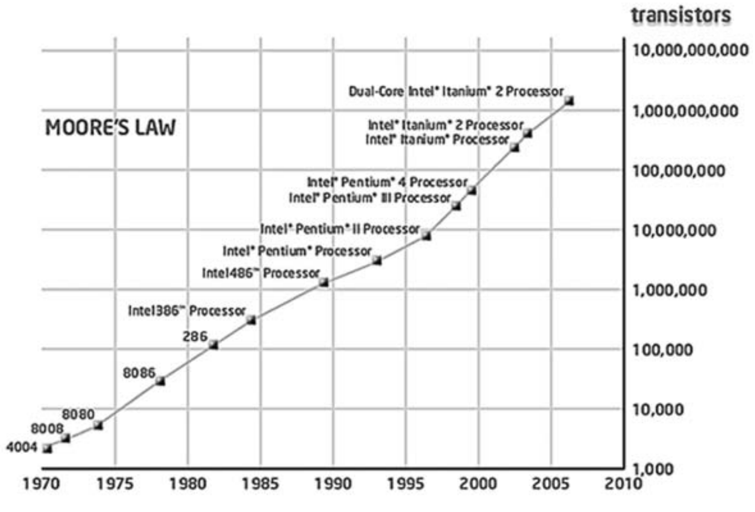


Figure 15.29. Evolution of transistor density

15.7.1. THE LIFE CYCLE OF SCIENTIFIC COMPUTING CODES

Let us consider the effects of Moore's Law on the lifecycle of typical large-scale scientific computing codes. The lifecycle of these codes may be subdivided into the following stages:

- conception;
- demonstration/proof of concept;
- production code;
- widespread use and acceptance;
- commodity tool;
- embedding.

In the *conceptual* stage, the basic purpose of the code is defined, the physics to be simulated identified and proper algorithms are selected and coded. The many possible algorithms are compared, and the best is kept. A run during this stage may take weeks or months to complete. A few of these runs may even form the core of a PhD thesis.

The *demonstration* stage consists of several large-scale runs that are compared to experiments or analytical solutions. As before, a run during this stage may take weeks or months to complete. Typically, during this stage the relevant time-consuming parts of the code are optimized for speed.

Once the basic code is shown to be useful, it may be adopted for *production* runs. This implies extensive benchmarking for relevant applications, quality assurance, bookkeeping of versions, manuals, seminars, etc. For commercial software, this phase is also referred to as *industrialization* of a code. It is typically driven by highly specialized projects that qualify the code for a particular class of simulations, e.g. air conditioning or external aerodynamics of cars.

If the code is successful and can provide a simulation capability not offered by competitors, the fourth phase, i.e. *widespread* use and acceptance, will follow naturally. An important shift is then observed: the 'missionary phase' (why do we need this capability?) suddenly transitions into a 'business as usual phase' (how could we ever design anything without this capability?). The code becomes an indispensable tool in industrial research, development, design and analysis. It forms part of the widely accepted body of 'best practices' and is regarded as commercial off the shelf (COTS) technology.

One can envision a fifth phase, where the code is *embedded* into a larger module, e.g. a control device that 'calculates on the fly' based on measurement input. The technology embodied by the code has then become part of the common knowledge and the source is freely available.

The time from conception to widespread use can span more than two decades. During this time, computing power will have increased by a factor of 1:10 000. Moreover, during a decade, algorithmic advances and better coding will improve performance by at least another factor of 1:10. Let us consider the role of parallel computing in light of these advances.

During the *demonstration* stage, runs may take weeks or months to complete on the largest machine available at the time. This places heavy emphasis on parallelization. Given that optimal performance is key, and massive parallelism seems the only possible way of solving the problem, ***distributed memory parallelism*** on $O(10^3)$ processors is perhaps the only possible choice. The figure of $O(10^3)$ processors is derived from experience: even as a high-end user with sometimes highly visible projects the author has never been able to obtain a larger number of processors with consistent availability in the last two decades. Moreover, no improvement is foreseeable in the future. The main reason lies in the usage dynamics of large-scale computers: once online, a large audience requests time on it, thereby limiting the maximum number of processors available on a regular basis for production runs.

Once the code reaches *production* status, a shift in emphasis becomes apparent. More and more ‘options’ are demanded, and these have to be implemented in a timely manner. Another five years have passed and by this time, processors have become faster (and memory has increased) by a further factor of 1:10, implying that the same run that used to take $O(10^3)$ processors can now be run on $O(10^2)$ processors. Given this relatively small number of processors, and the time constraints for new options/variants, ***shared memory parallelism*** becomes the most attractive option.

The *widespread* acceptance of a successful code will only accentuate the emphasis on quick implementation of options and user-specific demands. Widespread acceptance also implies that the code will no longer run exclusively on supercomputers, but will migrate to high-end servers and ultimately PCs. The code has now been in production for at least 5 years, implying that computing power has increased again by another factor of 1:10. The same run that used to take $O(10^3)$ processors in the demonstration stage can now be run using $O(10^1)$ processors, and soon will be within reach of $O(1)$ processors. Given that user-specific demands dominate at this stage, and that the developers are now catering to a large user base working mostly on low-end machines, ***parallelization diminishes in importance***, even to the point of completely disappearing as an issue. As parallelization implies extra time devoted to coding, thereby hindering fast code development, it may be removed from consideration at this stage.

One could consider a fifth phase, 20 years into the life of the code. The code has become an indispensable commodity tool in the design and analysis process, and is run thousands of times per day. Each of these runs is part of a stochastic analysis or optimization loop, and is performed on a commodity chip-based, uni-processor machine. Moore’s Law has effectively ***removed parallelism*** from the code.

Figure 15.30 summarizes the life cycle of typical scientific computing codes.

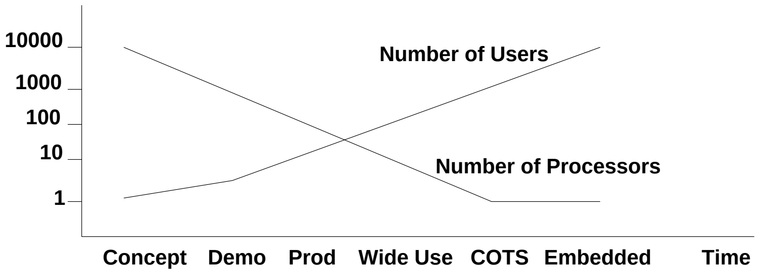


Figure 15.30. Life cycle of scientific computing codes

15.7.2. EXAMPLES

Let us consider two examples where the life cycle of codes described above has become apparent.

15.7.2.1. External missile aerodynamics

The first example considers aerodynamic force and moment predictions for missiles. World-wide, approximately 100 new missiles or variations thereof appear every year. In order to assess their flight characteristics, the complete force and moment data for the expected flight envelope must be obtained. Simulations of this type based on the Euler equations require approximately $O(10^6–10^7)$ elements, special limiters for supersonic flows, semi-empirical estimation of viscous effects and numerous specific options such as transpiration boundary conditions, modelling of control surfaces, etc. The first demonstration/feasibility studies took place in the early 1980s. At that time, it took the fastest production machine of the day (Cray-XMP) a night to compute such flows. The codes used were based on structured grids (Chakravarthy and Szema (1987)) as the available memory was small compared to the number of gridpoints. The increase of memory, together with the development of codes based on unstructured (Mavriplis (1991b), Luo *et al.* (1994)) or adaptive Cartesian grids (Melton *et al.* (1993), Aftosmis *et al.* (2000)) as well as faster, more robust solvers (Luo *et al.* (1998)) allowed for a high degree of automation. At present, external missile aerodynamics can be accomplished on a PC in less than an hour, and runs are carried out daily by the thousands for envelope scoping and simulator input on PC clusters (Robinson (2002)). Figure 15.31 shows an example.

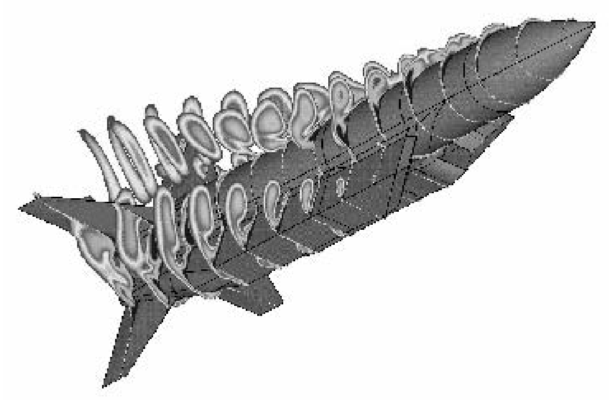


Figure 15.31. External missile aerodynamics

15.7.2.2. Blast simulations

The second example considers pressure loading predictions for blasts. Simulations of this type based on the Euler equations require approximately $O(10^6–10^8)$ elements, special limiters for transient shocks, and numerous specific options such as links to damage prediction post-processors. The first demonstration/feasibility studies took place in the early 1990s (Baum and Löhner (1991), Baum *et al*. (1993, 1995, 1996)). At that time, it took the fastest available machine (Cray-C90 with special memory) several days to compute such flows. The increase of processing power via shared memory machines during the past decade has allowed for a considerable increase in problem size, physical realism via coupled CFD/CSD runs (Löhner and Ramamurti (1995), Baum *et al*. (2003)) and a high degree of automation. At present, blast predictions with $O(2 \times 10^6)$ elements can be carried out on a PC in a matter of hours (Löhner *et al*. (2004c)), and runs are carried out daily by the hundreds for maximum possible damage assessment on networks of PCs. Figure 15.32 shows the results of such a prediction based on genetic algorithms for a typical city environment (Togashi *et al*. (2005)). Each dot represents an end-to-end run (grid generation of approximately 1.5 million tetrahedra, blast simulation with advanced CFD solver, damage evaluation), which takes approximately 4 hours on a high-end PC. The scale denotes the estimated damage produced by the blast at the given point. This particular run was done on a network of PCs and is typical of the migration of high-end applications to PCs due to Moore's Law.

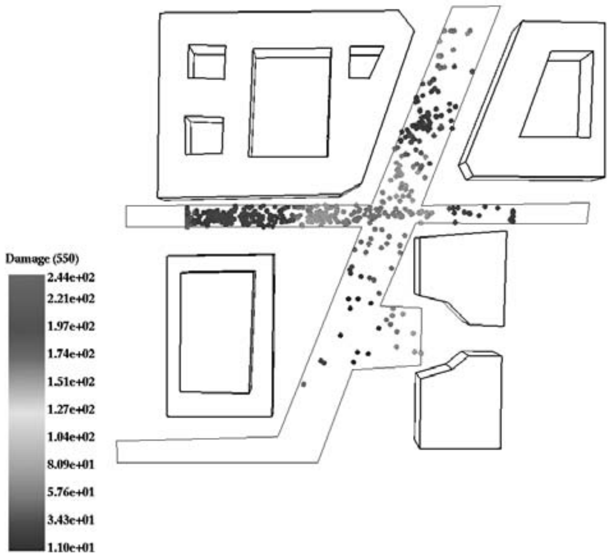


Figure 15.32. Maximum possible damage assessment for inner city

15.7.3. THE CONSEQUENCES OF MOORE'S LAW

The statement that parallel computing diminishes in importance as codes mature is predicated on two assumptions:

- the doubling of computing power every 18 months will continue;

- the total number of operations required to solve the class of problems the code was designed for has an asymptotic (finite) value.

The second assumption may seem the most difficult to accept. After all, a natural side effect of increased computing power has been the increase in problem size (grid points, material models, time of integration, etc.). However, for any class of problem there is an intrinsic limit for the problem size, given by the physical approximation employed. Beyond a certain point, the physical approximation does not yield any more information. Therefore, we may have to accept that parallel computing diminishes in importance as a code matures.

This last conclusion does not in any way diminish the overall significance of parallel computing. Parallel computing is an *enabling* technology of vital importance for the development of new high-end applications. Without it, innovation would seriously suffer.

On the other hand, without Moore's Law many new code developments would appear as unjustified. If computing time does not decrease in the future, the range of applications would soon be exhausted. CFD developers worldwide have always assumed subconsciously Moore's Law when developing improved CFD algorithms and techniques.

16 SPACE-MARCHING AND DEACTIVATION

For several important classes of problems, the propagation behaviour inherent in the PDEs being solved can be exploited, leading to considerable savings in CPU requirements. Examples where this propagation behaviour can lead to faster algorithms include:

- *detonation*: no change to the flowfield occurs ahead of the denotation wave;
- *supersonic flows*: a change of the flowfield can only be influenced by upstream events, but never by downstream disturbances; and
- *scalar transport*: a change of the transported variable can only occur in the downstream region, and only if a gradient in the transported variable or a source is present.

The present chapter shows how to combine physics and data structures to arrive at faster solutions. Heavy emphasis is placed on space-marching, where these techniques have reached considerable maturity. However, the concepts covered are generally applicable.

16.1. Space-marching

One of the most efficient ways of computing supersonic flowfields is via so-called space-marching techniques. These techniques make use of the fact that in a supersonic flowfield no information can travel upstream. Starting from the upstream boundary, the solution is obtained by marching in the downstream direction, obtaining the solution for the next downstream plane (for structured (Kutler (1973), Schiff and Steger (1979), Chakravarthy and Szema (1987), Matus and Bender (1990), Lawrence *et al.* (1991)) or semi-structured (McGrory *et al.* (1991), Soltani *et al.* (1993)) grids), subregion (Soltani *et al.* (1993), Nakahashi and Saitoh (1996), Morino and Nakahashi (1999)) or block. In the following, we will denote as a *subregion* a narrow band of elements, and by a *block* a larger region of elements (e.g. one-fifth of the mesh). The updating procedure is repeated until the whole field has been covered, yielding the desired solution.

In order to estimate the possible savings in CPU requirements, let us consider a steady-state run. Using local timesteps, it will take an explicit scheme approximately $O(n_s)$ steps to converge, where n_s is the number of points in the streamwise direction. The total number of operations will therefore be $O(n_t \cdot n_s^2)$, where n_t is the average number of points in the transverse planes. Using space-marching, we have, ideally, $O(1)$ steps per active domain, implying a total work of $O(n_t \cdot n_s)$. The gain in performance could therefore approach $O(1 : n_s)$ for large n_s. Such gains are seldomly realized in practice, but it is not uncommon to see gains in excess of 1:10.

Of the many possible variants, the space-marching procedure proposed by Nakahashi and Saitoh (1996) appears as the most general, and is treated here in detail. The method can be used with any explicit time-marching procedure, it allows for embedded subsonic regions and is well suited for unstructured grids, enabling a maximum of geometrical flexibility. The method works with a subregion concept (see Figure 16.1). The flowfield is only updated in the so-called active domain. Once the residual has fallen below a preset tolerance, the active domain is shifted. Should subsonic pockets appear in the flowfield, the active domain is changed appropriately.

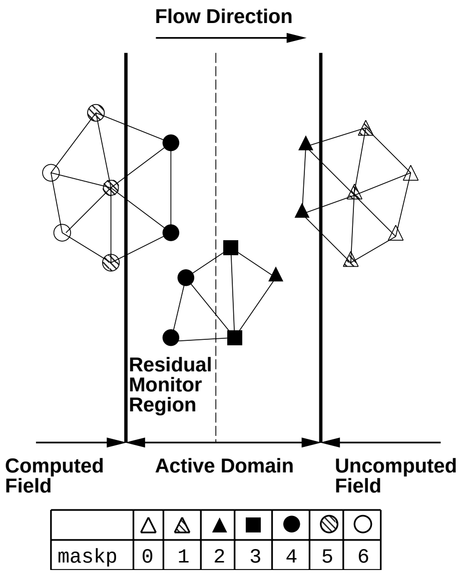


Figure 16.1. Masking of points

In the following, we consider computational aspects of Nakahashi and Saitoh's space-marching scheme and a blocking scheme in order to make them as robust and efficient as possible without a major change in existing codes. The techniques are considered in the following order: masking of edges and points, renumbering of points and edges, grouping to avoid memory contention, extrapolation of the solution for new active points, treatment of subsonic pockets, proper measures for convergence, the use of space-marching within implicit, time-accurate solvers for supersonic flows and macro-blocking.

16.1.1. MASKING OF POINTS AND EDGES

As seen in the previous chapters, any timestepping scheme requires the evaluation of fluxes, residuals, etc. These operations typically fall into two categories:

(a) *point Loops*, which are of the form

```
do ipoin=1,npoin
   do work on the point level
enddo
```

(b) *edge loops*, which are of the form

```
do iedge=1,nedge
   gather point information
   do work on the edge level
   scatter-add edge results to points
enddo
```

The first loop is typical of unknown updates in multistage Runge–Kutta schemes, initialization of residuals or other point sums, pressure, speed of sound evaluations, etc. The second loop is typical of flux summations, artificial viscosity contributions, gradient calculations and the evaluation of the allowable timestep. For cell-based schemes, point loops are replaced by cell loops and edge loops are replaced by face loops. However, the nature of these loops remains the same. The bulk of the computational effort of any scheme is usually carried out in loops of the second type.

In order to decide where to update the solution, points and edges need to be classified or 'masked'. Many options are possible here, and we follow the notation proposed by Nakahashi and Saitoh (1996) (see Figure 16.1):

`maskp=0:`	point in downstream, uncomputed field;
`maskp=1:`	point in downstream, uncomputed field, connected to active domain;
`maskp=2:`	point in active domain;
`maskp=3:`	point of `maskp=2,` with connection to points of `maskp=4;`
`maskp=4:`	point in the residual-monitor subregion of the active domain;
`maskp=5:`	point in the upstream computed field, with connection to active domain;
`maskp=6:`	point in the upstream computed field.

The edges for which work has to be carried out then comprise all those for which at least one of the endpoints satisfies `0<maskp<6`. These active edges are marked as `maske=1`, while all others are marked as `maske=0`.

The easiest way to convert a time-marching code into a space- or domain-marching code is by rewriting the point- and edge loops as follows.

Loop 1a:

```
do ipoin=1,npoin
   if(maskp(ipoin).gt.0. and .maskp(ipoin).lt.6) then
      do work on the point level
   endif
enddo
```

Loop 2a:

```
do  iedge=1,nedge
    if(maske(iedge).eq.1) then
       gather point information
       do work on the edge level
       scatter-add edge results to points
    endif
enddo
```

For typical aerodynamic configurations, resolution of geometrical detail and flow features will dictate the regions with smaller elements. In order to be as efficient as possible, the region being updated at any given time should be chosen as small as possible. This implies that, in regions of large elements, there may exist edges that connect points marked as `maskp=4` to points marked as `maskp=0`. In order to leave at least one layer of points in the safety region, a pass over the edges is performed, setting the downstream point to `maskp=4` for edges with point markings `maskp=2,0`.

16.1.2. RENUMBERING OF POINTS AND EDGES

For a typical space-marching problem, a large percentage of points in Loop 1a will not satisfy the `if`-statement, leading to unnecessary work. Renumbering the points according to the marching direction has the twofold advantage of a reduction in cache-misses, and the possibility to bound the active point region locally. Defining

`npami:`	the minimum point number in the active region,
`npamx:`	the maximum point number in the active region,
`npdmi:`	the minimum point number touched by active edges,
`npdmx:`	the maximum point number touched by active edges,

Loop 1a may now be rewritten as follows.

Loop 1b:

```
do ipoin=npami,npamx
   if(maskp(ipoin).gt.0. and .maskp(ipoin).lt.6) then
      do work on the point level
   endif
enddo
```

For the initialization of residuals, the range would become `npdmi,npdmx`. In this way, the number of unnecessary `if`-statements is reduced significantly, leading to considerable gains in performance.

As was the case with points, a large number of redundant `if`-tests may be avoided by renumbering the edges according to the minimum point number. Such a renumbering also reduces cache-misses, a major consideration for RISC-based machines. Defining

`neami:`	The minimum active edge number;
`neamx:`	The maximum active edge number;

Loop 2a may now be rewritten as follows.

Loop 2b:

```
do iedge=neami,neamx
   if(maske(iedge).eq.1) then
      gather point information
      do work on the edge level
      scatter-add edge results to points
   endif
enddo
```

16.1.3. GROUPING TO AVOID MEMORY CONTENTION

In order to achieve pipelining or vectorization, memory contention must be avoided. The enforcement of pipelining or vectorization is carried out using a compiler directive, as Loop 2b, which becomes an inner loop, and still offers the possibility of memory contention. In this case, we have the following:

Loop 2c:

```
do  ipass=1,npass
    nedg0=edpas(ipass)+1
    nedg1=edpas(ipass+1)
c$dir ivdep                                        ! Pipelining directive
    do iedge=nedg0,nedg1
       if(maske(iedge).eq.1) then
          gather point information
          do work on the edge level
          scatter-add edge results to points
       endif
    enddo
enddo
```

It is clear that in order to avoid memory contention, for each of the groups of edges (inner loop), none of the corresponding points may be accessed more than once. Given that in order to achieve good pipelining performance on current RISC chips a relatively short vector length of 16 is sufficient, one can simply start from the edge-renumbering obtained before, and renumber the edges further into groups of 16, while avoiding memory contention (see Chapter 15). For CRAYs and NECs, the vector length chosen ranges from 64 to 256.

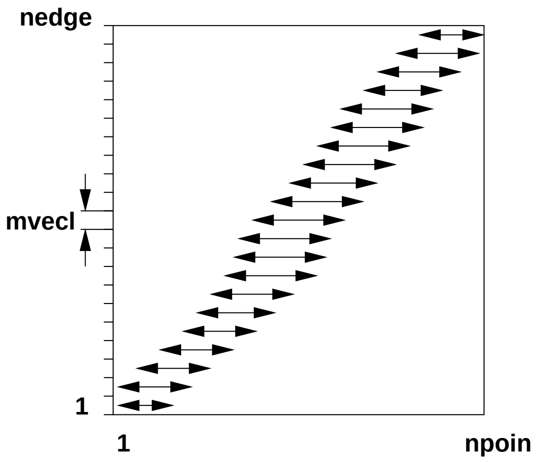


Figure 16.2. Near-optimal point-range access of edge groups

The loop structure is shown schematically in Figure 16.2. One is now in a position to remove the `if`-statement from the innermost loop, situating it outside. The inactive edge groups are marked, e.g. `edpas(ipass)<0`. This results in the following.

Loop 2d:

```
do  ipass=1,npass
    nedg0=abs(edpas(ipass))+1
    nedg1=  edpas(ipass+1)
    if(nedg1.gt.0) then
c$dir ivdep                                      ! Pipelining directive
       do iedge=nedg0,nedg1
          gather point information
          do work on the edge level
          scatter-add edge results to points
       enddo
    endif
enddo
```

Observe that the innermost loop is the same as that for the original time-marching scheme. The change has occurred at the outer loop level, leading to a considerable reduction of unnecessary `if`-tests, at the expense of a slightly larger number of active edges, as well as a larger bandwidth of active points.

16.1.4. EXTRAPOLATION OF THE SOLUTION

As the solution progresses downstream, a new set of points becomes active, implying that the unknowns are allowed to change there. The simplest way to proceed for these points is to start from whatever values were set at the beginning and iterate onwards. In many cases, a better way to proceed is to extrapolate the solution from the closest point that was active during the previous timestep. This extrapolation is carried out by looping over the new active edges, identifying those that have one point with known solution and one with unknown solution, and setting the values of the latter from the former. This procedure may be refined by keeping track of the alignment of the edges with the flow direction and extrapolating from the point that is most aligned with the flow direction (see Figure 16.3). Given that more than one layer of points may be added when a new region is updated, an open loop over the new edges is performed, until no new active points with unknown solution are left. This extrapolation of the unknowns can significantly reduce the number of iterations required for convergence, making it well worth the effort.

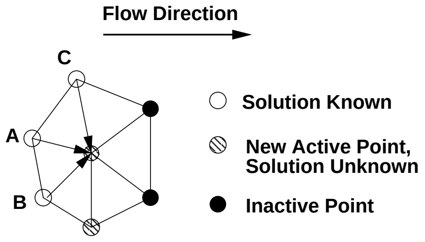


Figure 16.3. Extrapolation of the solution

16.1.5. TREATMENT OF SUBSONIC POCKETS

The appearance of subsonic pockets in a flowfield implies that the active region must be extended properly to encompass it completely. Only then can the ‘upstream-only’ argument be applied.

In this case, the planes are simply shifted upstream and downstream in order to satisfy this criterion. For small subsonic pockets, which are typical of hypersonic airplanes, a more expedient way to proceed is shown in Figure 16.4. The spatial extent of subsonic points upstream and downstream of the active region is obtained, leading to the ‘conical’ regions C_u, C_d. All edges and points in these regions are then marked as `lpoin(ipoin)=2,4`, respectively. All other steps are kept as before. Subsonic pockets tend to change during the initial formation and subsequent iterations. In order to avoid the repeated marking of points and edges, the conical regions are extended somewhat. Typical values of this ‘safety zone’ are $s = 0.1$–0.2 `dxsaf`.

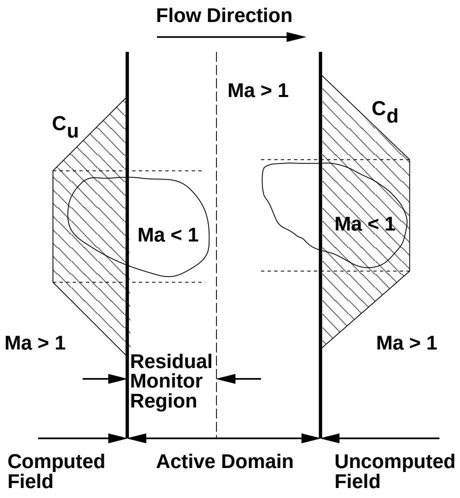


Figure 16.4. Treatment of subsonic pockets

16.1.6. MEASURING CONVERGENCE

Any iterative procedure requires a criterion to decide when the solution has converged. If we write an explicit time-marching scheme as

$$\mathbf{M}_l \Delta \mathbf{u}^n = \mathbf{R}^n, \tag{16.1}$$

where R^n and $\Delta\mathbf{u}^n$ denote the residual and change of unknowns for the nth timestep, respectively, and $\mathbf{M}_l$ is the lumped mass matrix, the convergence criterion most commonly used is some global integral of the form

$$r^n = \int_\Omega |\Delta\mathbf{u}^n| \, d\Omega \approx \sum_i \mathbf{M}_l^i |\Delta\hat{\mathbf{u}}_i^n|. \tag{16.2}$$

r^n is compared to r^1 and, if the ratio of these numbers is sufficiently small, the solution is assumed converged. Given that for the present space-marching procedure the residual is only measured in a small but varying region, this criterion is unsatisfactory. One must therefore attempt to derive different criteria to measure convergence. Clearly the solution may be assumed to be converged if the maximum change in the unknowns over the points of the mesh has decreased sufficiently:

$$\max_i(|\Delta\hat{\mathbf{u}}_i^n|) < \epsilon_0. \tag{16.3}$$

In order to take away the dimensionality of this criterion, one should divide by an average or maximum of the unknowns over the domain:

$$\frac{\max_i(|\Delta\hat{\mathbf{u}}_i^n|)}{\max_i(\hat{\mathbf{u}}_i)} < \epsilon_1. \tag{16.4}$$

The quantity $\Delta\mathbf{u}$ depends directly on the timestep, which is influenced by the Courant number selected. This dependence may be removed by dividing by the Courant number *CFL* as follows:

$$\frac{\max_i(|\Delta\hat{\mathbf{u}}_i^n|)}{CFL\,\max_i(\hat{\mathbf{u}}_i)} < \epsilon_2. \tag{16.5}$$

This convergence criterion has been found to be quite reliable, and has been used for the examples shown below. When shocks are present in the flowfield, some of the limiters will tend to switch back and forth for points close to the shocks. This implies that, after the residual has dropped to a certain level, no further decrease is possible. This ‘floating’ or ‘hanging up’ of the residuals has been observed and documented extensively. In order not to iterate *ad infinitum*, the residuals are monitored over several steps. If no meaningful decrease or increase is discerned, the spatial domain is updated once a preset number of iterations in the current domain has been exceeded.

16.1.7. APPLICATION TO TRANSIENT PROBLEMS

The simulation of vehicles manoeuvering in supersonic and hypersonic flows, or aeroelastic problems in this flight regime, require a time-accurate flow solver. If an implicit scheme of the form (e.g. Alonso *et al.* (1995))

$$\tfrac{3}{2}\mathbf{M}_l^{n+1}\mathbf{u}^{n+1} - 2\mathbf{M}_l^n\mathbf{u}^n + \tfrac{1}{2}\mathbf{M}_l^{n-1}\mathbf{u}^{n-1} = \Delta t\mathbf{R}^{n+1} \tag{16.6}$$

is solved using a pseudo-timestep approach as

$$\frac{d}{d\tau}\mathbf{u} + \mathbf{R}^* = 0, \tag{16.7}$$

where

$$\mathbf{R}^* = \tfrac{3}{2}\mathbf{M}_l^{n+1}\mathbf{u}^{n+1} - 2\mathbf{M}_l^n\mathbf{u}^n + \tfrac{1}{2}\mathbf{M}_l^{n-1}\mathbf{u}^{n-1} - \Delta t\mathbf{R}^{n+1}, \tag{16.8}$$

an efficient method of solving this pseudo-timestep system for supersonic and hypersonic flow problems is via space-marching.

16.1.8. MACRO-BLOCKING

The ability of the space-marching technique described to treat possible subsonic pockets requires the availability of the whole mesh during the solution process. This may present a problem for applications requiring very large meshes, where machine memory constraints can easily be reached. If the extent of possible subsonic pockets is known – a situation that is quite common – the computational domain can be subdivided into subregions, and each can be updated in turn. In order to minimize user intervention, the mesh is first generated for the complete domain. This has the advantage that the CAD data does not need to be modified. This large mesh is then subdivided into blocks. Since memory overhead associated with splitting programs is almost an order of magnitude less than that of a flow code, even large meshes can be split without reaching the memory constraints the flow code would have for the smaller sub-domain grids.

Once the solution is converged in an upstream domain, the solution is extrapolated to the next downstream domain. It is obvious that boundary conditions for the points in the upstream 'plane' have to be assigned the 'no allowed change' boundary condition of supersonic inflow. For the limiting procedures embedded in most supersonic flow solvers, gradient information is required at points. In order to preserve full second-order accuracy across the blocks, and to minimize the differences between the uni-domain and blocking solutions, the second layer of upstream points is also assigned a 'no allowed change' boundary condition (see Figure 16.5). At the same time, the overlap region between blocks is extended by one layer. Figure 16.6 shows the solutions obtained for a 15° ramp and $Ma_\infty = 3$ employing the usual uni-domain scheme, and two blocking solutions with one and two layers of overlap respectively. As one can see, the differences between the solutions are small, and are barely discernable for two layers of overlap.

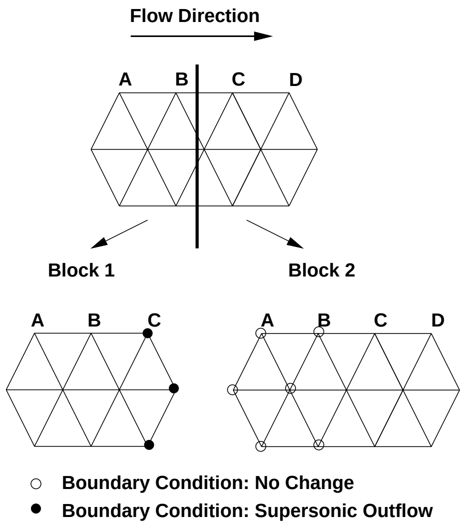

Flow Direction
A
B
C
D
Block 1
Block 2
A
B
C
A
B
C
D
○ Boundary Condition: No Change
● Boundary Condition: Supersonic Outflow

Figure 16.5. Macro-blocking with two layers of overlap

Within each sub-domain, space-marching may be employed. In this way, the solution is obtained in an almost optimal way, minimizing both CPU and memory requirements.

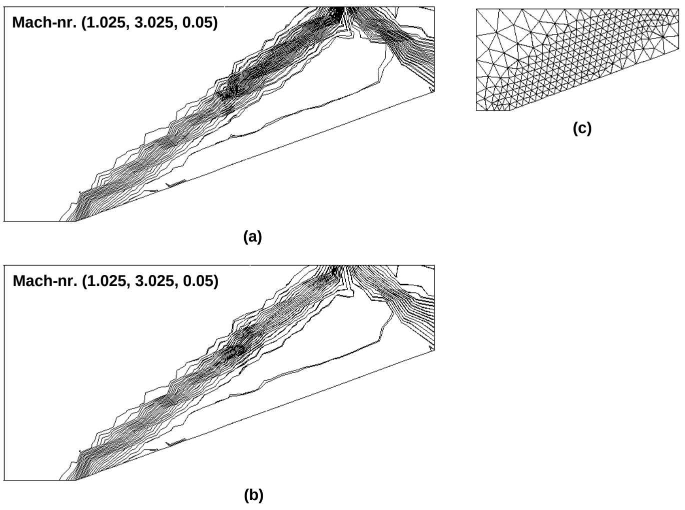


Figure 16.6. Macro-blocking: (a) one layer of overlap; (b) two layers of overlap; (c) mesh used

16.1.9. EXAMPLES FOR SPACE-MARCHING AND BLOCKING

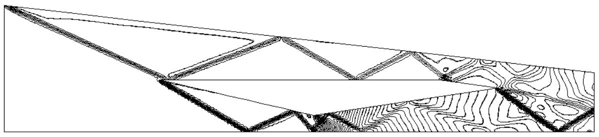
The use of space-marching and macro-blocking is exemplified on several examples. In all of these examples the Euler equations are solved, i.e. no viscous effects are considered.

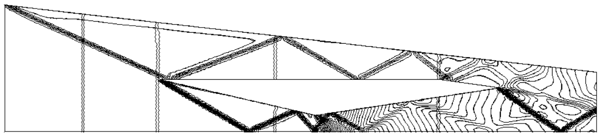
16.1.9.1. Supersonic inlet flow

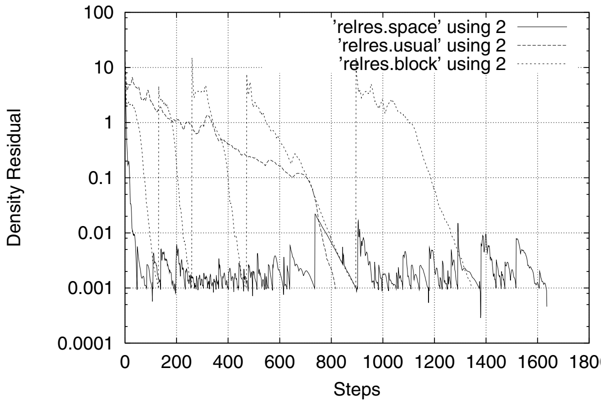
This internal supersonic flow case, taken from Nakahashi and Saitoh (1996), represents part of a scramjet intake. The total length of the device is $l = 8.0$, and the element size was set uniformly throughout the domain to $\delta = 0.03$. The cross-section definition is shown in Figure 16.7(a). Although this is a 2-D problem, it is run using a 3-D code. The inlet Mach number was set to $Ma = 3.0$. The mesh (not shown, since it would blacken the domain) consisted of 540 000 elements and 106 000 points, of which 30 000 were boundary points. The flow solver is a second-order Roe solver that uses MUSCL reconstruction with pointwise gradients and a vanAlbada limiter on conserved quantities. A three-stage scheme with a Courant number of `CFL=1.0` and three residual smoothing passes were employed. The convergence criterion was set to $\epsilon_2 = 10^{-3}$.

The Mach numbers obtained for the space-marching and the usual time-marching procedure are superimposed in Figure 16.7(a). As one can see, these contours are almost indistinguishable, indicating that the convergence criterion used is proper. The solution was also obtained using blocking. The individual blocking domains are shown for clarity in Figure 16.7(b). The five blocks consisted of 109 000, 103 000, 126 000, 113 000 and 119 000 elements, respectively. The convergence history for all three cases – usual timestepping, space-marching and blocking – is summarized in Figure 16.7(c). Table 16.1 summarizes the CPU requirements on an SGI R10000 processor for different marching and safety-zone sizes, as well as for usual time-marching and blocking.

The first observation is that although this represents an ideal case for space-marching, the speedup observed is not spectacular, but worth the effort. The second observation is that the


(a)


(b)



(c)

Figure 16.7. Mach number: (a) usual versus space-marching, min = 0.825, max = 3.000, incr = 0.05; (b) usual versus blocking min = 0.825, max = 3.000, incr = 0.05; (c) convergence history for inlet

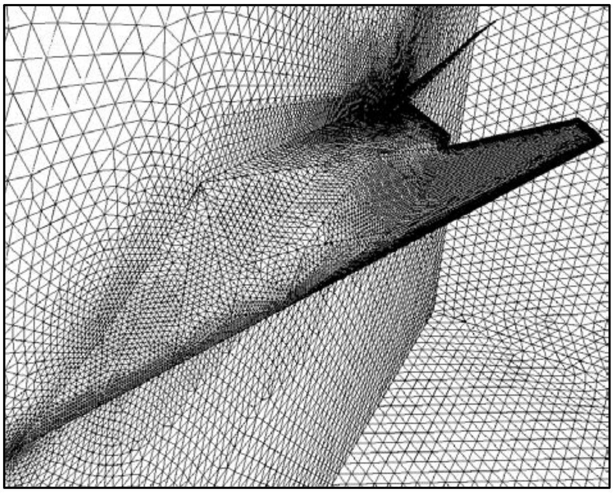
Table 16.1. Timings for inlet (540 000 elements)

`dxmar`	`dxsaf`	CPU (min)	Speedup
Usual		400	1.00
0.05	0.20	160	2.50
0.10	0.40	88	4.54
0.10	0.60	66	6.06
Block		140	2.85

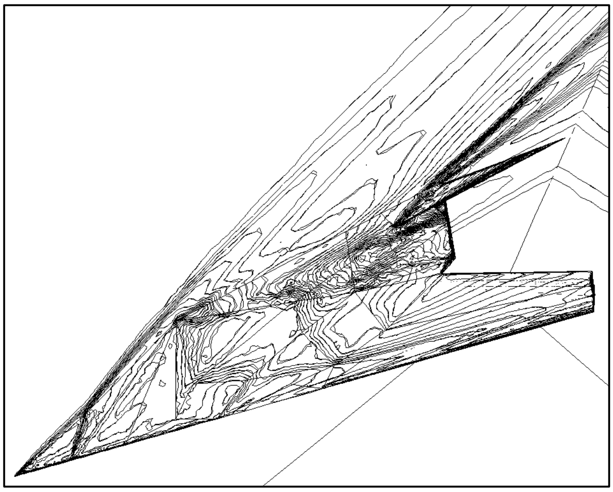
speedup is sensitive to the safety zone ahead of the converged solution. This is a user-defined parameter, and a convincing way of choosing automatically this distance has so far remained elusive.

16.1.9.2. F117

As a second case, we consider the external supersonic flow at $Ma = 4.0$ and $\alpha = 4.0°$ angles of attack over an F117-like geometry. The total length of the airplane is $l = 200.0$.



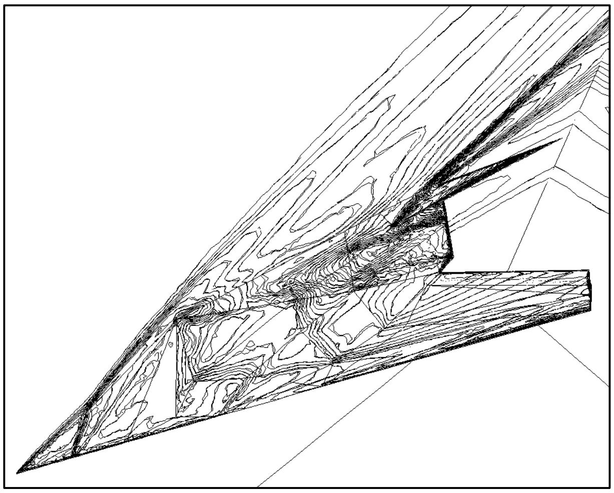
(a)



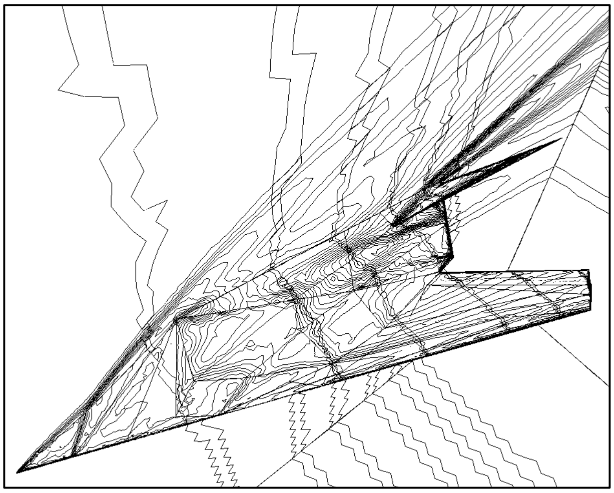
(b)

Figure 16.8. (a) F117 surface mesh; (b) Mach number: usual versus space-marching, min = 0.55, max = 6.50, incr = 0.1; (c) Mach number: usual versus blocking, min = 0.55, max = 6.50, incr = 0.1; (d) Mach number contours for blocking solution, min = 0.55, max = 6.50, incr = 0.1; (e) convergence history for F117

The unstructured surface mesh is shown in Figure 16.8(a). Smaller elements were placed close to the airplane in order to account for flow gradients. The mesh consisted of 2 056 000 elements and 367 000 points, of which 35 000 were boundary points. As in the previous case, the flow solver is a second-order Roe solver that uses MUSCL reconstruction with pointwise gradients and a vanAlbada limiter on conserved quantities. A three-stage scheme with a Courant number of `CFL=1.0` and three residual smoothing passes were employed.



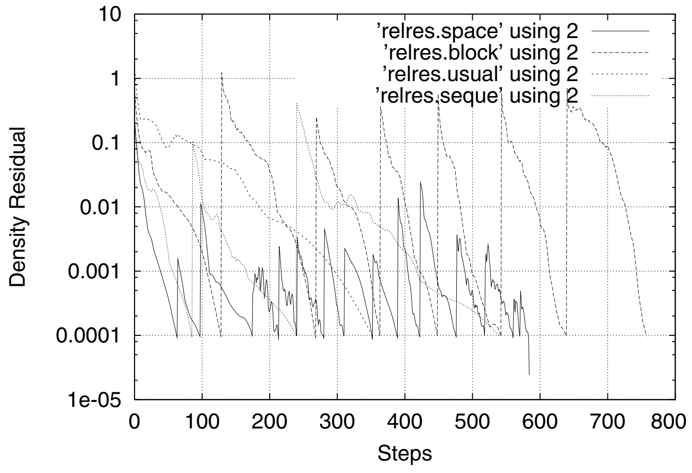
(c)



(d)

Figure 16.8. Continued

The convergence criterion was set to $\epsilon_2 = 10^{-4}$. The Mach numbers obtained for the space-marching, usual time-marching and blocking procedures are superimposed in Figures 16.8(b) and (c). The individual blocking domains are shown for clarity in Figure 16.8(d). The seven blocks consisted of 357 000, 323 000, 296 000, 348 000, 361 000, 386 000 and 477 000 elements, respectively. As before, these contours are almost indistinguishable, indicating a proper level of convergence. Table 16.2 summarizes the CPU requirements on an SGI R10000 processor for different marching and safety-zone sizes, macro-blocking, as well as for usual time-marching and grid sequencing. The coarser meshes consisted of 281 000 and



(e)

Figure 16.8. Continued

Table 16.2. Timings for F117 (543 000 tetrahedra, 106 000 points)

`dxmar`	`dxsaf`	CPU (min)	Speedup
Usual		611	1.00
Seque		518	1.17
10	30	227	2.69
20	30	218	2.80
Block	1	260	2.35

42 000 elements respectively. The residual curves for the three different cases are compared in Figure 16.8(e). As one can see, grid sequencing only provides a marginal performance improvement for this case. Space-marching is faster than blocking, although not by a large margin.

16.1.9.3. Supersonic duct flow with moving parts

This case simulates the same geometry and inflow conditions as the first case. The center-piece, however, is allowed to move in a periodic way as follows:

$$x_c = x_c^0 + a \cdot \sin(\omega t). \tag{16.9}$$

For the case considered here, $x_c^0 = 4.2$, $a = 0.2$ and $\omega = 0.05$. The mesh employed is the same as that of the first example, and the same applies to the spatial discretization part of the flow solver used. The implicit timestepping scheme given by (16.6) is used to advance the solution in time, and the pseudo-timestepping of the residuals, given by (16.7), is carried out using space-marching, with the convergence criterion set to $\epsilon_2 = 10^{-3}$. Each period was discretized by 40 timesteps, yielding a timestep $\Delta t = \pi$ and a Courant number of approximately 300. The number of space-marching steps required for each implicit timestep was approximately 600, i.e. similar to one steady-state run. Figure 16.9 shows the Mach number distribution at different times during the third cycle.

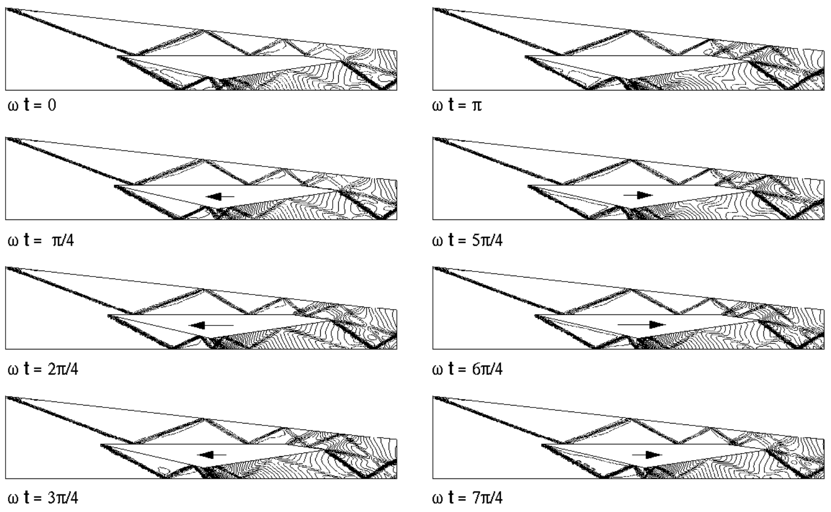


Figure 16.9. Inlet flowfield with oscillating inner part, Mach number: min = 0.875, max = 3.000 incr = 0.05

16.2. Deactivation

The space-marching procedure described above achieved CPU gains by working only on a subset of the complete mesh. The same idea can be used advantageously in other situations, leading to the general concept of deactivation. Two classes of problems where deactivation has been used extensively are point detonation simulations (Löhner *et al.* (1999b)) and scalar transport (Löhner and Camelli (2004)). In order to mask points and edges (faces, elements) in an optimal way, and to avoid any unnecessary `if`-statements, the points are renumbered according to the distance from the origin of the explosion, or in the streamline direction. This idea can be extended to multiple explosion origins and to recirculation zones, although in these cases sub-optimal performance is to be expected. For the case of explosions, only the points and edges that can have been reached by the explosion are updated. Similarly, for scalar transport problems described by the classic advection-diffusion equation

$$c_{,t} + \mathbf{v} \cdot \nabla c = \nabla k \nabla c + S, \tag{16.10}$$

where c, $\mathbf{v}$, k and S denote the concentration, velocity, diffusivity of the medium and source term, respectively, a change in c can only occur in those regions where

$$|S| > 0, \quad |\nabla c| > 0. \tag{16.11}$$

For the typical contaminant transport problem, the extent of the regions where $|S| > 0$ is very small. In most of the regions that lie upwind of a source, $|\nabla c| = 0$. This implies that in a considerable portion of the computational domain no contaminant will be present, i.e. $c = 0$. As stated before, the basic idea of deactivation is to identify the regions where no change in c can occur, and to avoid unnecessary work in them.

The marking of deactive regions is accomplished in two loops over the elements. The first loop identifies in which elements sources are active, i.e. where $|S| > 0$. The second loop identifies in which elements/edges a change in the values of the unknowns occurs, i.e. where $\max(c_e) - \min(c_e) > \epsilon_u$, with ϵ_u a preset, very small tolerance. Once these active elements/edges have been identified, they are surrounded by additional layers of elements which are also marked as active. This 'safety' ring is added so that changes in neighbouring elements can occur, and so that the test for deactivation does not have to be performed at every timestep. Typically, four to five layers of elements/edges are added. From the list of active elements, the list of active points is obtained. The addition of elements to form the 'safety' ring can be done in a variety of ways. If the list of elements surrounding elements or elements surrounding points is available, only local operations are required to add new elements. If these lists are not present, one can simply perform loops over the edges, marking points, until the number of 'safety layers' has been reached. In either case, it is found that the cost of these marking operations is small compared to the advancement of the transport equation.

16.2.1. EXAMPLES OF DYNAMIC DEACTIVATION

The use of dynamic deactivation is exemplified on several examples.

16.2.1.1. Generic weapon fragmentation

The first case considered is a generic weapon fragmentation, and forms part of a fully coupled CFD/CSD run (Baum *et al.* (1999)). The structural elements are assumed to fail once the average strain in an element exceeds 60%. At the beginning, the fluid domain consists of two separate regions. These regions connect as soon as fragmentation starts. In order to handle narrow gaps during the break-up process, the failed structural elements are shrunk by a fraction of their size. This alleviates the timestep constraints imposed by small elements without affecting the overall accuracy. The final breakup leads to approximately 1200 objects in the flowfield. Figure 16.10 shows the fluid pressure, the mesh velocity and the surface velocity of the structure at three different times during the simulation. The edges and points are checked every 5 to 10 timesteps and activated accordingly. The deactivation technique leads to considerable savings in CPU at the beginning of a run, where the timestep is very small and the zone affected by the explosion only comprises a small percentage of the mesh. Typical meshes for this simulation were of the order of 8.0 million tetrahedra, and the simulations required of the order of 50 hours on the SGI Origin2000 running on 32 processors.

16.2.1.2. Subway station

The second example considers the dispersion of an instantaneous release in the side platform of a generic subway station, and is taken from Löhner and Camelli (2004). The geometry is shown in Figure 16.11(a).

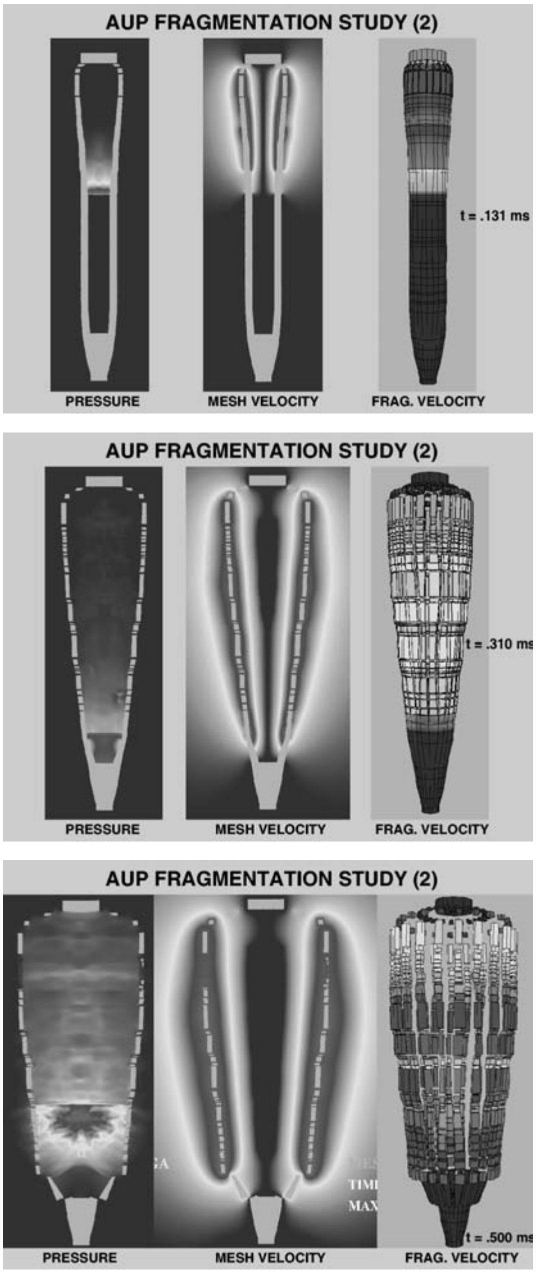


Figure 16.10. Pressure, mesh and fragment velocities at three different times

A time-dependent inflow is applied on one of the end sides:

$$v(t) = b(t - 60)^3 e^{-a(t-60)} + v_0$$

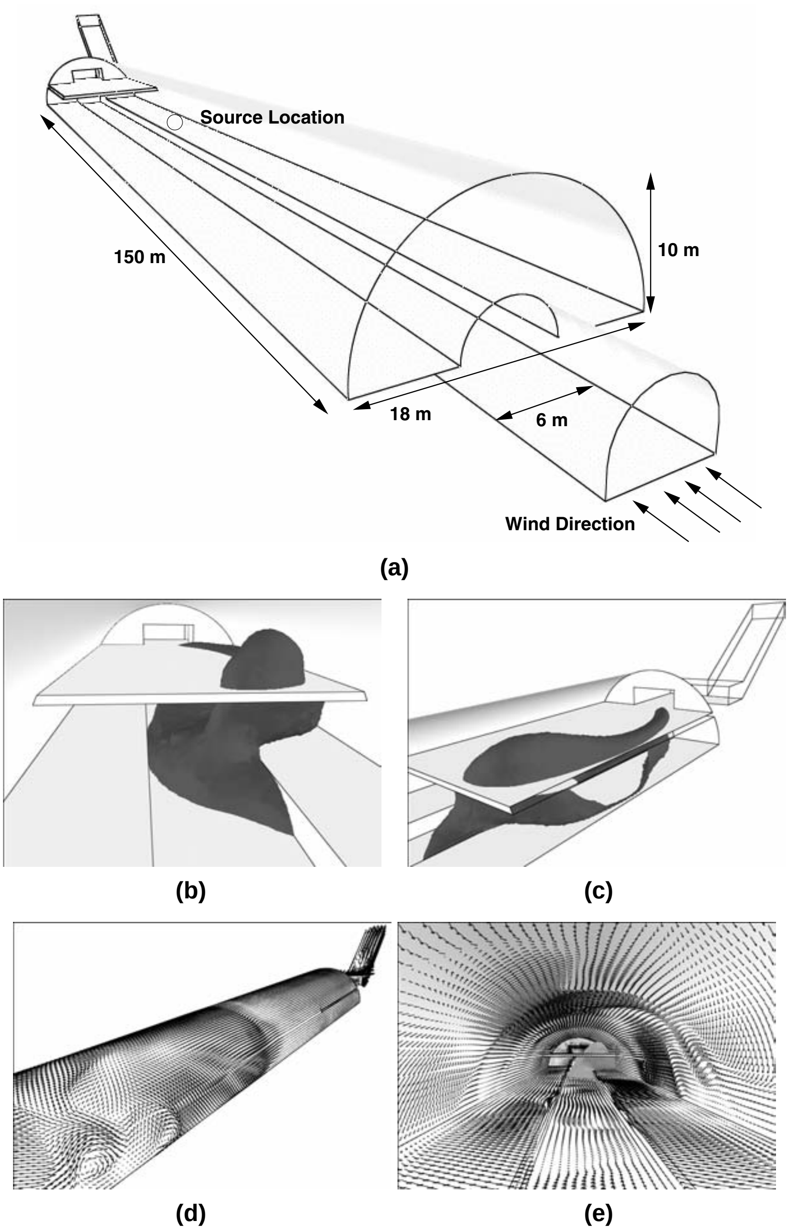


Figure 16.11. (a) Problem definition; (b), (c) iso-surface of concentration $c = 0.0001$; (d), (e) surface velocities

where $b = 0.46$ m/s, $a = 0.5$ 1/s, and $v_0 = 0.4$ m/s. This inflow velocity corresponds approximately to the velocities measured at a New York City subway station (Pflistch *et al.* (2000)). The Smagorinsky model with the Law of the Wall was used for this example. The volume grid has 730 000 elements and 144 000 points. The dispersion simulation was performed using a three-stage Runge–Kutta scheme with a Courant number of $C = 0.6$. The dispersion calculation was run for 485 s of real time (corresponding to the time of a train entering, halting, exiting the station and the time for the next train to arrive) on a workstation with the following characteristics: Dec Alpha chip running at 0.67 GHz, 4 Gbyte of RAM, Linux operating system, Compaq compiler. Figures 16.11(b)–(e) show the resulting iso-surface of concentration level $c = 0.0001$, as well as the surface velocities for time $t = 485$ s. Note the transient nature of the flowfield, which is reflected in the presence of many vortices.

Deactivation checks were performed every 5 timesteps. The tolerance for deactivation was set to $\epsilon_u = 10^{-3}$. The usual run (i.e. without deactivation) took `T=5,296 sec`, whereas the run with deactivation took `T=526 sec`, i.e. a saving in excess of 1:10. This large reduction in computing time was due to two factors: the elements with the most constraining timestep are located at the entry and exit sections, and the concentration cloud only reaches this zone very late in the run (or not at all); furthermore, as this is an instantaneous release, the region of elements where concentration is present in meaningful values is always very small as compared to the overall domain. Speedups of this magnitude have enabled the use of 3-D CFD runs to assess the maximum possible damage of contaminant release events (Camelli (2004)) and the best possible placement of sensors (Löhner (2005)).

17 OVERLAPPING GRIDS

As seen in previous chapters, the last two decades have witnessed the appearance of a number of numerical techniques to handle problems with complex geometries and/or moving bodies. The main elements of any comprehensive capability to solve this class of problems are:

- automatic grid generation;
- solvers for moving grids/boundaries; and
- treatment of grids surrounding moving bodies.

Three leading techniques to resolve problems of this kind are as follows.

- *Solvers based on unstructured, moving (ALE) grids*. These guarantee conservation, a smooth variation of grid size, but incur extra costs due to mesh movement/smoothing techniques and remeshing, and may have locally reduced accuracy due to distorted elements; for some examples, see Löhner (1990), Baum *et al.* (1995b), Löhner *et al.* (1999b) and Sharov *et al.* (2000).
- *Solvers based on overlapping grids*. These do not require any remeshing, can use fast mesh movement techniques, allow for independent component/body gridding, but suffer from loss of conservation, incur extra costs due to interpolation and may have locally reduced accuracy due to drastic grid size variation; for some examples, see Steger *et al.* (1983), Benek *et al.* (1985), Buning *et al.* (1988), Dougherty and Kuan (1989), Meakin and Suhs (1989), Meakin (1993), Nirschl *et al.* (1994), Meakin (1997), Rogers *et al.* (1998) and Regnström *et al.* (2000), Togashi *et al.* (2000), Löhner (2001), Togashi *et al.* (2006a,b).
- *Solvers based on adaptive embedded grids*. These do not require remeshing or mesh movement, but extra effort is necessary to detect the location of boundaries with respect to the mesh, to refine/coarsen the mesh as bodies move, and to treat properly the small elements that appear when moving boundaries cut through the mesh. RANS gridding presents another set of as yet unresolved problems; for some examples, see Quirk (1994), Karman (1995), Pember *et al.* (1995), Landsberg and Boris (1997), Aftosmis and Melton (1997), Aftosmis *et al.* (2000), LeVeque and Calhoun (2001), del Pino and Pironneau (2001), Peskin (2002) and Löhner (2004).

A large body of work exists on overlapping grids; in fact, a whole series of conferences is devoted to the subject (Overset (2002–2006)). In a series of papers, Nakahashi and co-workers (Nakahashi *et al.* (1999), Togashi *et al.* (2000)) developed and applied overlapping grids with unstructured grids, showing good performance for the overall scheme. A general 'distance to wall' criterion was used to assign which points should be interpolated between grids. For each grid system, the distance to the body walls was computed. Any point from a given grid falling into the element of another grid was interpolated whenever its distance to wall was larger than that of the points of the other grid. The interpolation information was obtained from a nearest-neighbour technique.

In the following, the main elements required for any solver based on overlapping grids will be discussed: interpolation criteria between overlapping grids, proper values of dominant mesh criteria for background grids (those grids that have no bodies or walls assigned to them) and the external boundaries of embedded grids that require interpolation and lie outside the computational domain, interpolation techniques for static and dynamic data, treatment of grids that are partially outside the flowfield, and the changes required for flow solvers.

17.1. Interpolation criteria

Given a set of overlapping grids, a criterion must be formulated in order to select which points will interpolate information from one grid to another in the overlapping regions. Criteria that are used extensively include:

- a fixed number of layers around bodies or inside external domains;
- the distance to the wall δ_w, whereby points closer to a body interpolate to those that are farther away (Nakahashi *et al.* (1999)).

The accuracy of any (convergent) CFD solver will increase as the element size decreases. Therefore, it seems natural to propose as the 'dominant mesh criterion' the element size h itself. In this way, the point that is surrounded by the smallest elements is the one on which the solution is evaluated. All that is required is the average element size at points. This criterion will not work for grids with uniform element size. For this reason, a better choice for the 'dominant mesh criterion' is the product of distance to wall and mesh size $\delta \cdot h$.

In what follows, we will denote by s the generalized 'distance to wall' criterion given by

$$s = \delta^p \cdot h^q, \tag{17.1}$$

which includes all of the criteria described above:

(a) $p = 1, q = 0$ for distance to wall;

(b) $p = 0, q = 1$ for element size;

(c) $p = 1, q = 1$ for a combination of both criteria.

The generalized 'distance to wall' described above requires the determination of the closest distance to the bodies/walls for each gridpoint. A brute force calculation of the shortest distance from any given point to the wall faces would require $O(N_p \cdot N_b)$, where N_p denotes the number of points and N_b the number of wall boundary points. This is clearly unacceptable for 3-D problems, where $N_b \approx N_p^{2/3}$. A near-optimal way to construct the desired distance function was described at the end of Chapter 2 (see section 2.7). It uses a combination of heap lists, the point surrounding point linked list `psup1, psup2`, the list of faces on the boundary `bface`, as well as the faces surrounding points linked list `fsup1, fsup2`. The key idea is to move from the surfaces into the domain, obtaining the closest points/faces to any given point from previously computed, near-optimal starting locations. Except for points in the middle of a domain, the algorithm will yield the correct distance to walls with an algorithmic complexity of $O(N_p \log(N_p))$, which is clearly much superior to a brute force approach.

17.2. External boundaries and domains

In many instances, the bodies in the flowfield are surrounded by their own grids, which in turn are embedded into an external 'background grid' that defines the farfield flow conditions (see Figure 17.1). In order to define a proper 'dominant mesh criterion' for the external grid, the maximum value of s (see (17.1) above) is computed for the whole domain. All points belonging to the external grid are then assigned a value of $s_{\text{ext}} = 2s_{\max}$. In order to force an interpolation for the points belonging to external boundaries of embedded grids (as well as the points of two layers adjacent to the external boundaries), the value of s for these points is set to an artificially high value, e.g. $s_{\text{out}} = 10s_{\max}$.

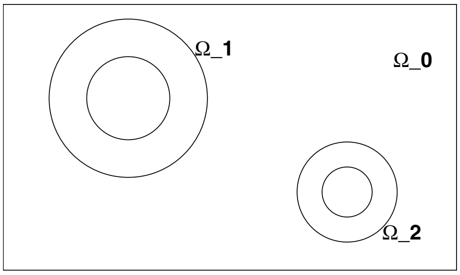


Figure 17.1. External boundaries

17.3. Interpolation: initialization

Any overlapping grid technique requires information transfer between grids, i.e. some form of interpolation. This implies that for each point we need to see if there exists an element in a different grid from which the flowfield variables should be interpolated. A fast, parallel way of determining the required interpolation information is obtained by reversing the question, i.e. by determining which points of a different grid fall into any given element. A near-optimal way to construct the desired interpolation information requires a combination of:

- a marker for the domain number of each point; and
- an octree of all gridpoints.

The algorithm consists of two parts, which may be summarized as follows (see Figure 17.2).

Part 1: Initialization

- Mark the domain number of each gridpoint;
- Construct an octree for all gridpoints;

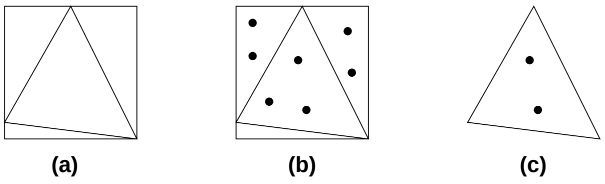


Figure 17.2. Interpolation: (a) bounding box; (b) points (octree); (c) retain points – different domain and inside element

Part 2: Interpolation

```
- do: loop over the elements
-   Obtain the bounding box for the element;
-   From the octree, obtain the points in the bounding box;
-   Retain only the points that:
        Correspond to other domains/grids;
        Are located inside the element;
        Have a value of $s$ larger than the one in the element;
- enddo
```

Observe that the loop over the elements does not contain any recursions or memory contention, implying that it may be parallelized in a straightforward manner.

The number of elements to be tested can, in many cases, be reduced significantly by presorting the elements that should be tested. A typical situation is a large background grid with one or two objects inside. There is no need to test elements of this background grid that are outside the bounding boxes of the interior meshes. A more sophisticated filter can be obtained via bins. The idea is to retain only elements that fall into bins covered by elements of more than one mesh/zone. This very effective filter may be summarized as follows.

Part 1: Initialization

```
- Determine the outer bounding box of all grids;
- Determine the number of subdivisions (bins) in $x$, $y$, $z$;
- Set lbins(1:nbins)=0
- Set lelem(1:nelem)=0
```

Part 2: First pass over the elements

```
- do ielem=1,nelem                              ! Loop over the elements
-   Obtain the material/mesh/zone nr. of the element: iemat;
-   Obtain the bounding box for the element;
-   Obtain the bins covered by the bounding box;
  - do: Loop over the bins covered:
      If: lbins(ibin).eq.0: lbins(ibin)=iemat;
      If: lbins(ibin).gt.0. and .lbins(ibin).ne.iemat:
          lbins(ibin)=-1;
  - enddo
- enddo
```

Part 3: Second pass over the elements

```
- do ielem=1,nelem                                    ! Loop over the elements
- Obtain the bounding box for the element;
- Obtain the bins covered by the bounding box;
   - do: Loop over the bins covered:
      If: lbins(ibin).eq.-1: lelem(ielem)=1;
   - enddo
- enddo
```

At the end of this second pass over the elements, all the elements that cover bins covered by elements of more than one mesh/zone will be marked as `lelem(ielem)=1`. Only these elements are used to obtain the interpolation information between grids.

The use of any 'dominant mesh criterion' s can produce points that interpolate to and are interpolated from other grids. A schematic of such a situation is shown in Figure 17.3 for a 1-D domain. Point A of the background grid Ω_0 is used for interpolating to the outer points of the interior grid Ω_1, but in turn passes the distance criterion and is interpolated from Ω_1 as well. Although clearly dangerous, it is found that in most cases this will not produce invalid results. The points that are interpolated from other grids and interpolate to other grids in turn are eliminated from the list of interpolating points. For more than two grids, care has to be taken in order to eliminate pairwise grid correspondences. For example, it could happen that point i of grid A is interpolated from grid B but interpolates to grid C. In such cases, the points have to be kept in their respective lists.

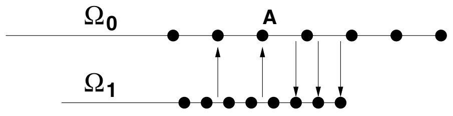


Figure 17.3. Interpolating interpolation points

17.4. Treatment of domains that are partially outside

In some instances, the external boundaries of a region associated with a body will lie outside the 'background grid'. A typical case is illustrated in Figure 17.4, where a portion of the boundary of body 1 with domain Ω_1 lies outside the confines of the channel Ω_0.

The interpolation to these points from Ω_0 will be impossible. In order to treat cases such as this, the points that could not find host elements on a different grid are examined further. Any nearest-neighbour of these points that has not found a host is marked as belonging to an exterior boundary, and the distance to body is set to an accordingly high value. Thereafter, the interpolation between grids is attempted once more. This procedure, shown schematically in Figure 17.5, is repeated until no further points can be added to the list of exterior points that have not found a host.

17.5. Removal of inactive regions

In most instances, the distance to wall criterion will produce a list of interpolation points that contains more points than required. A typical case is illustrated in Figure 17.6, where Ω_1, an

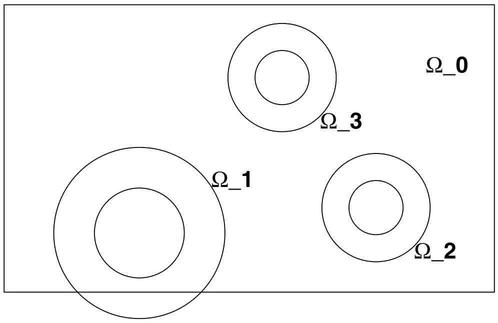


Figure 17.4. Non-inclusive domains

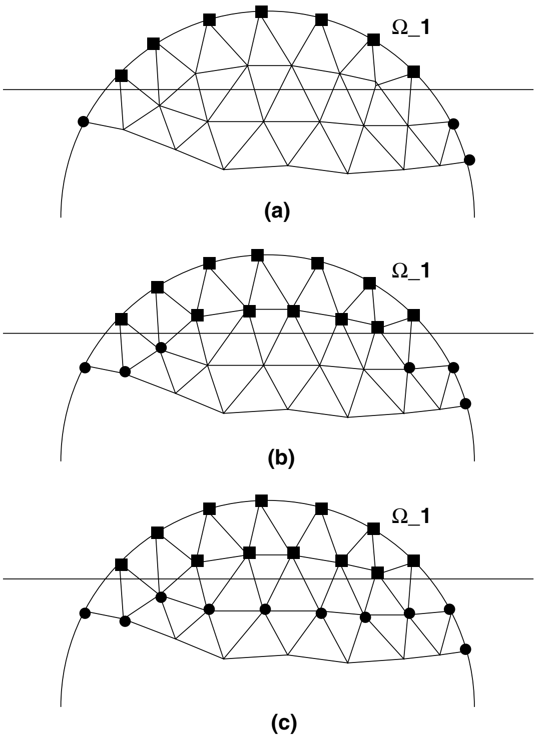
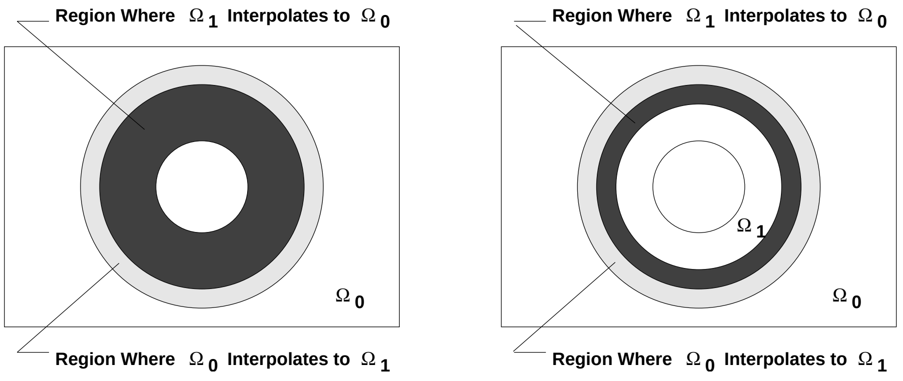


Figure 17.5. Treatment of external domains

internal region associated with a moving body, covers a substantial portion of the ‘background grid’ Ω_0. This implies that many points of Ω_0 will be included in the interpolation lists. Clearly, the interior region covering Ω_1 does not need to be updated. It is sufficient to interpolate a number of ‘layers’.



Region Where Ω_1 Interpolates to Ω_0
Ω_0
Region Where Ω_0 Interpolates to Ω_1
Region Where Ω_1 Interpolates to Ω_0
Ω_1
Ω_0
Region Where Ω_0 Interpolates to Ω_1

Figure 17.6. Removal of inactive inside regions

An algorithm that can remove these inactive interior regions may be summarized as follows:

```
- Initialize a point array lpoin(1:npoin)=1;
- For all interpolated points: Set lpoin(i)=0;
- do: Loop over the layers required
    - For each point: Obtain the maximum of lpoin and its neighbours;
- enddo
- Remove all interpolating points for which lpoin(i)=0.
```

17.6. Incremental interpolation

Although fast and parallel, the interpolation technique discussed above can be improved substantially for moving bodies. The key assumption made is that the moving bodies will only move a small number (one, possibly two or three) of nearest-neighbours during a timestep. In this case, the points surrounding currently interpolated points can be marked, and the distance to wall test to determine intergrid interpolation is only performed for this set of elements/points. Given that the octree for the points would have to be rebuilt every timestep for moving bodies, it is advisable to use the classic neighbour-to-neighbour element search shown schematically in Figure 17.7 and discussed in Chapter 13. The marking of points, as well as the interpolation, can again be parallelized without difficulties on shared-memory machines.

17.7. Changes to the flow solver

The changes required for any flow solver based on explicit timestepping or iterative implicit schemes are surprisingly modest. All that is required is a masking array over the points. For those points marked as inactive or as being interpolated, the increments in the solution are simply set to zero. The solution is interpolated at the end of each explicit timestep, or within each iteration for LU-SGS-GMRES (Luo *et al.* (1998)) or PCG (Ramamurti and Löhner (1996)) solvers.

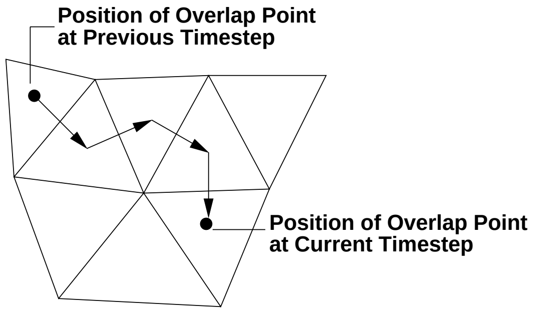

Position of Overlap Point at Previous Timestep
Position of Overlap Point at Current Timestep

Figure 17.7. Neighbour-to-neighbour element search

17.8. Examples

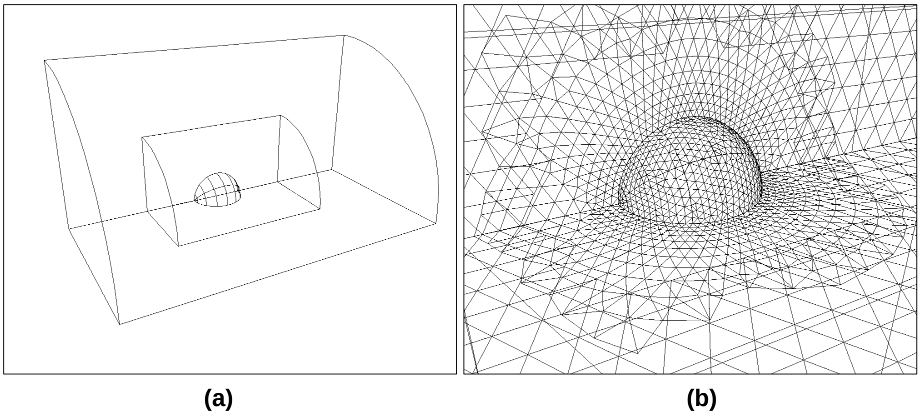
In order to demonstrate the viability of overlapping grid techniques, we consider a series of compressible and incompressible flow problems. The first examples are included to show that the results of overlapping grids are almost identical to those of single grids.

17.8.1. SPHERE IN CHANNEL (COMPRESSIBLE EULER)

The first case consists of a sphere in a channel. The equations solved are the compressible Euler equations. The incoming flow is at $Ma = 0.67$. The CAD data is shown in Figure 17.8(a), where the boundaries of the different regions are clearly visible. The surface grids and the solution obtained are shown in Figures 17.8(b) and (c). One can see the smooth transition of contour lines in the overlap region. The solution obtained for a single grid case with similar mesh size is also plotted, and as one can see, the differences are negligible. The convergence rate for both cases is summarized in Figure 17.8(d). Observe that the convergence rate is essentially the same for the single grid and overlap grid case.

17.8.2. SPHERE IN SHEAR FLOW (INCOMPRESSIBLE NAVIER–STOKES)

The second case consists of a sphere in shear flow. The equations solved are the incompressible Navier–Stokes equations. The Reynolds number based on the sphere diameter and the incoming flow is $Re = 10$. The CAD data is shown in Figure 17.9(a), where the boundaries of the different regions are clearly visible. The grid in a cut plane, as well as the pressure and absolute values of the velocity for the overlapping and single grid case are given in Figures 17.9(b)–(d). The overlapping and single grids had 327 961 and 272 434 elements, respectively. The convergence rates for both cases are summarized in Figure 17.9(e). As in the previous case, the differences between the overlapping and single grid cases are negligible. The body forces for the single and overlapping grid case were $f_x = 2.2283$, $f_y = 0.3176$ and $f_x = 2.2115$, $f_y = 0.3289$, respectively, i.e. 0.7% in x and 3.4% in y.



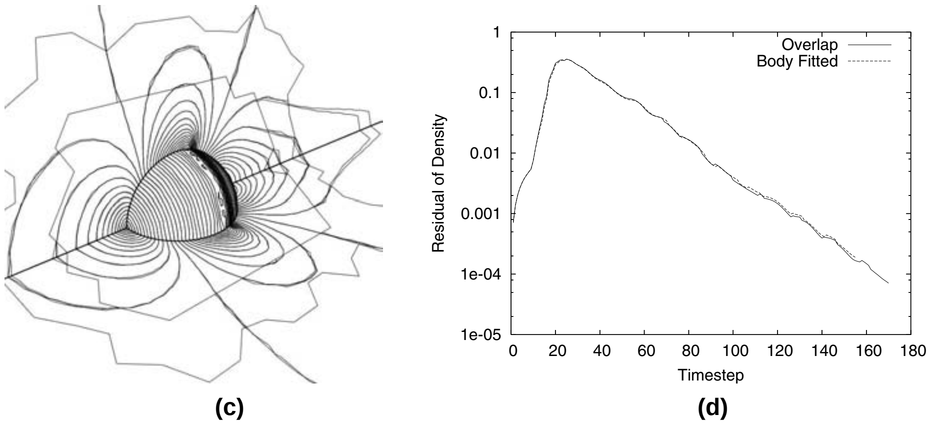
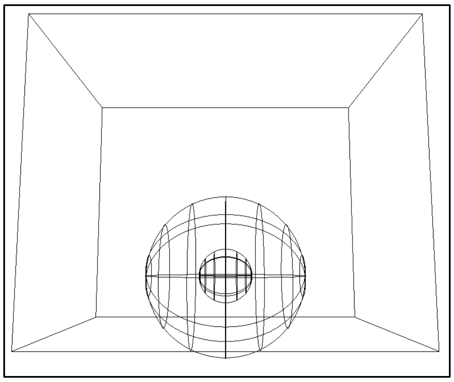


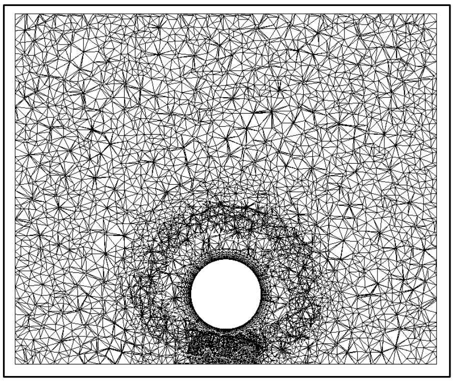
Figure 17.8. Sphere in channel: (a) CAD data; (b) surface grids; (c) Mach-number; (d) convergence rates

17.8.3. SPINNING MISSILE

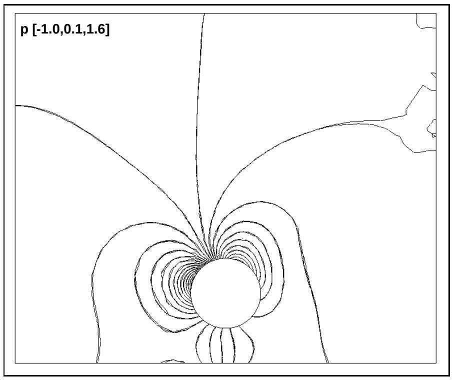
This is a typical case where overlapping grids can be used advantageously as compared to the traditional moving/deforming mesh with regridding approach. A missile flying at $Ma = 2.5$ at an angle of attack of $\alpha = 5°$ spins around its axis. The surface definition of the computational domains is shown in Figure 17.10(a). The inner grid is associated with the missile and rotates rigidly with the prescribed spin rate. The outer grid is fixed and serves as background grid to impose the farfield boundary conditions. Once the initial interpolation points have been determined, the interpolation parameters for subsequent interpolations on the moved grids are obtained using the incremental near-neighbour search technique. The compressible Euler equations are integrated using a Crank–Nicholson timestepping scheme. At each timestep, the resulting algebraic system is solved using the LU-SGS–GMRES technique (Luo *et al.* (1998, 1999), Sharov *et al.* (2000b)). Figures 17.10(b) and (c) show the grid and pressures in the plane $z = 0$ for a particular time. The overlap regions are clearly visible. Note that one can barely see any discrepancy in the contour lines across the overlap region; neither are there any spurious reflections or anomalies in the shocks across the overlap region.



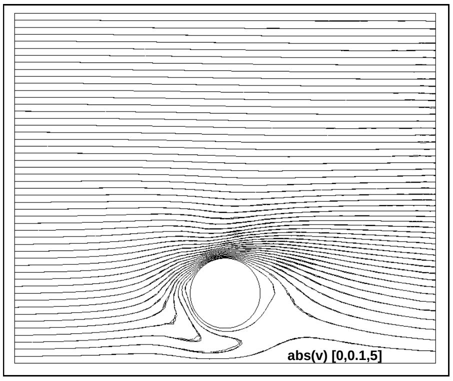
(a)



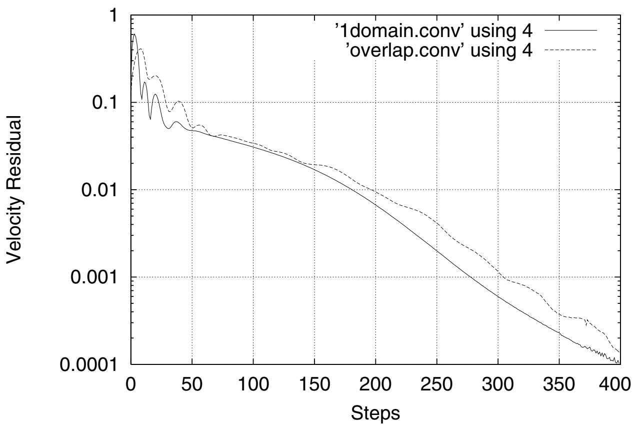
(b)



(c)

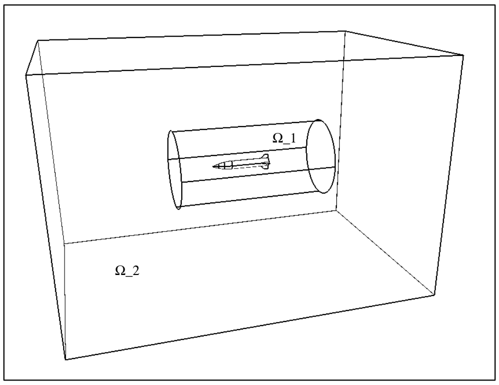


(d)

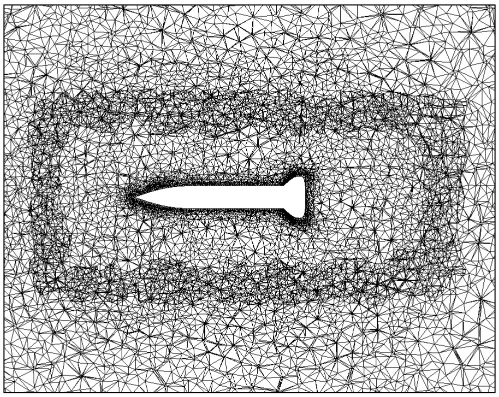


(e)

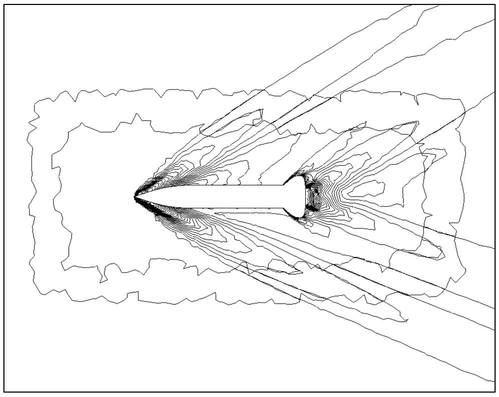
Figure 17.9. Sphere in shear flow: (a) CAD data; (b) triangulation for cut plane; (c) pressure in cut plane; (d) Abs(vel) in cut plane; (e) convergence rates



(a)



(b)



(c)

Figure 17.10. Spinning missile: (a) computational domain; (b) plane $z = 0$, mesh; (c) plane $z = 0$, pressure contours

18 EMBEDDED AND IMMERSED GRID TECHNIQUES

As seen throughout the previous chapters, the numerical solution of PDEs is usually accomplished by performing a spatial and temporal discretization with subsequent solution of a large algebraic system of equations. The transition from an arbitrary surface description to a proper mesh still represents a difficult task. This is particularly so when the surface description is based on data that does not originate from CAD systems, such as data from remote sensing, medical imaging or fluid–structure interaction problems. Considering the rapid advance of computer power, together with the perceived maturity of field solvers, an automatic transition from an arbitrary surface description to a mesh becomes mandatory.

So far, we have only considered grids that are body-conforming, i.e. grids where the external mesh faces match up with the surface (body surfaces, external surfaces, etc.) of the domain. This chapter will consider the case when elements and points do not match up perfectly with the body. Solvers or methods that employ these non-body-conforming grids are known by a variety of names: embedded mesh, fictitious domain, immersed boundary, immersed body, Cartesian method, etc. The key idea is to place the computational domain inside a large mesh (typically a regular parallelepiped), with special treatment of the elements and points close to the surfaces and/or inside the bodies. If we consider the general case of moving or deforming surfaces with topology change, as the mesh is not body-conforming, it does not have to move. Hence, the PDEs describing the flow can be left in the simpler Eulerian frame of reference even for moving surfaces. At every timestep, the elements/edges/points close to and/or inside the embedded/immersed surface are identified and proper boundary conditions are applied in their vicinity. While used extensively (Clarke *et al.* (1985), de Zeeuw and Powell (1991), Melton *et al.* (1993), Quirk (1994), Karman (1995), Pember *et al.* (1995), Landsberg and Boris (1997), Pember (1995), Roma (1995), Turek (1999), Lai and Peskin (2000), Aftosmis *et al.* (2000), Cortez and Minion (2000), LeVeque and Calhoun (2001), Dadone and Grossman (2002), Peskin (2002), Gilmanov *et al.* (2003), Löhner *et al.* (2004a,b), Gilmanov and Sotiropoulos (2005), Mittal and Iaccarino (2005), Deng *et al.* (2006)) this solution strategy also exhibits some shortcomings:

- the boundary, which, in the absence of field sources, has the most profound influence on the ensuing physics, is also the place where the worst elements/approximations are found;

- near the boundary, the embedding boundary conditions need to be applied, in many cases reducing the local order of approximation for the PDE;

- it is difficult to introduce stretched elements to resolve boundary layers;

- adaptivity is essential for most cases;
- for problems with moving boundaries the information required to build the proper boundary conditions for elements close to the surface or inside the bodies can take a considerable amount of time; and
- for fluid–structure interaction problems, obtaining the information required to transfer forces back to the structural surface can also take a considerable amount of time.

In nearly all cases reported to date, embedded or immersed boundary techniques were developed as a response to the treatment of problems with:

- ‘dirty geometries’ (Landsberg and Boris (1997), Aftosmis *et al.* (2000), Baum *et al.* (2003), Camelli and Löhner (2006));
- moving/sliding bodies with thin/vanishing gaps (Baum *et al.* (2003), vande Voorde *et al.* (2004)); and
- physics that can be handled with isotropic grids:
    - potential flow or Euler: Quirk (1994), Karman (1995), Pember *et al.* (1995), Landsberg and Boris (1997), Aftosmis *et al.* (2000), LeVeque and Calhoon (2001), Dadone and Grossman (2002), Baum *et al.* (2003),
    - Euler with boundary layer corrections: Aftosmis *et al.* (2006),
    - low-Reynolds-number laminar flow and/or large eddy simulation (LES): Angot *et al.* (1999), Turek (1999), Fadlun *et al.* (2000), Kim *et al.* (2001), Gilmanov *et al.* (2003), Balaras (2004), Gilmanov and Sotiropoulos (2005), Mittal and Iaccarino (2005), Cho *et al.* (2006), Yang and Balaras (2006).

The human cost of repairing bad input data sets can be overwhelming in many cases. For some cases, it may even constitute an impossible task. Consider, as an example, the class of fluid–structure interaction problems where surfaces may rupture and self-contact due to blast damage (Baum *et al.* (1999), Löhner *et al.* (1999a), Baum *et al.* (2003)). The contact algorithms of most computational structural dynamics (CSD) codes are based on some form of spring analogy, and hence cannot ensure strict no-penetration during contact. As the surface is self-intersecting, it becomes impossible to generate a topologically consistent, body-fitted volume mesh. In such cases, embedded or immersed boundary techniques represent the only viable solution.

Two basic approaches have been proposed to modify field solvers in order to accommodate embedded surfaces or immersed bodies. They are based on either *kinetic* or *kinematic* boundary conditions near the surface or inside the bodies in the fluid. The first type applies an *equivalent balancing force* to the flowfield in order to achieve the kinematic boundary conditions required at the embedded surface or within the embedded domain (Goldstein *et al.* (1993), Mohd-Yusof (1997), Angot *et al.* (1999), Patankar *et al.* (2000), Fadlun *et al.* (2000), Kim *et al.* (2001), del Pino and Pironneau (2001), Peskin (2002), vande Voorde *et al.* (2004), Balaras (2004), Tyagi and Acharya (2005), Mittal and Balaras (2005), Cho *et al.* (2006), Yang and Iaccarino (2006)). The second approach is to apply *kinematic boundary conditions* at the nodes close to the embedded surface (Landsberg and Boris (1997), Aftosmis *et al.* (2000), Dadone and Grossman (2002), Löhner *et al.* (2004a,b)).

At first sight, it may appear somewhat contradictory to even consider embedded surface or immersed body techniques in the context of a general unstructured grid solver. Indeed, most of the work carried out to date was in conjunction with Cartesian solvers (Clarke *et al.* (1985), Tsuboi *et al.* (1991), de Zeeuw and Powell (1991), Melton *et al.* (1993), Karman (1995), Pember *et al.* (1995), Landsberg and Boris (1997), LeVeque and Calhoon (2001), Aftosmis *et al.* (2000), Dadone and Pironneau (2002), Gilmanov *et al.* (2003), Gilmanov and Sotiropoulos (2005), Mittal and Iaccarino (2005), Cho *et al.* (2006)), the argument being that flux evaluations could be optimized due to coordinate alignment and that so-called fast Poisson solvers (multigrid, fast Fourier transform) could easily be employed. However, the achievable gains of such coordinate alignment may be limited due to the following mitigating factors.

- For most of the high-resolution schemes the cost of limiting and the approximate Riemann solver far outweigh the cost of the few scalar products required for arbitrary edge orientation.
- The fact that any of these schemes (Cartesian, unstructured) requires mesh adaptation in order to be successful immediately implies the use of indirect addressing (and also removes the possibility of using solvers based on fast Fourier transforms); given current trends in microchip design, indirect addressing, present in both types of solvers, may outweigh all other factors.
- Three specialized (x, y, z) edge loops versus one general edge loop, and the associated data reorganization, implies an increase in software maintenance costs.

Indeed, empirical evidence from explicit compressible flow solvers indicates that the gains achievable when comparing general, edge-based, unstructured grid solvers versus optimized Cartesian solvers amount to no more than a factor of 1:4, and in most cases are in the range of 1:2 (Löhner *et al.* (2005)).

For a general unstructured grid solver, surface embedding represents just another addition to a toolbox of mesh handling techniques (mesh movement, overlapping grids, remeshing, h-refinement, deactivation, etc.), and one that allows the treatment of ‘dirty geometry’ problems with surprising ease, provided the problem is well represented with isotropic grids. It also allows for a combination of different surface treatment options. A good example where this has been used very effectively is the modelling of endovascular devices such as coils and stents (Cebral and Löhner (2005)). The arterial vessels are gridded using a body-fitted unstructured grid while the endovascular devices are treated via an embedded technique.

In what follows, we denote by CSD faces the surface of the computational domain (or surface) that is embedded. We implicitly assume that this information is given by a triangulation, which typically is obtained from a CAD package via STL (stereo lithography) files, remote sensing data, medical images or from a CSD code (hence the name) in coupled fluid–structure applications. For immersed body methods we assume that the embedded object is given by a tetrahedral mesh. Furthermore, we assume that in both cases, besides the connectivity and coordinate information, the velocity of the points is also given.

18.1. Kinetic treatment of embedded or immersed objects

As stated before, one way of treating embedded objects is via the addition of suitable force functions that let the fluid ‘feel’ the presence of the surface, and push away any fluid trying to penetrate the same. If we consider a rigid, closed body, as sketched in Figure 18.1, an obvious aim is to enforce, within the body, the condition $\mathbf{v} = \mathbf{w}_b$ (recall that rigid body motion is a solution of the Navier–Stokes equations). This may be accomplished by applying a force term of the form:

$$\mathbf{f} = -c_0(\mathbf{w}_b - \mathbf{v}) \tag{18.1}$$

for points that are inside (and perhaps just outside) of the body. This particular type of force function is known as the penalty force technique (Goldstein *et al.* (1993), Mohd-Yusof (1997), Angot *et al.* (1999), Fadlun *et al.* (2000), Kim *et al.* (2001)).

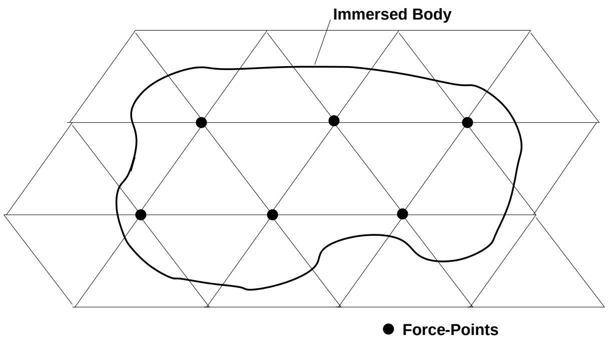


Figure 18.1. Kinetic treatment of embedded surfaces

Of course, other functional forms of $\mathbf{w}_b - \mathbf{v}$ are possible, e.g. the quadratic form

$$\mathbf{f} = -c_0|\mathbf{w}_b - \mathbf{v}|(\mathbf{w}_b - \mathbf{v}), \tag{18.2}$$

exponential forms, etc. The damping characteristics in time for the relaxation of a current velocity to the final state will vary, but the basic idea is the same. The advantage of the simple linear form given by (18.1) is that a point-implicit integration of the velocities is possible, i.e. the stiffness of the large coefficient c_0 can be removed with no discernable increase in operations (vande Voorde *et al.* (2004)). The main problem with the force fields given by (18.1) and (18.2) is the choice of the constants c_0. Values that are too low do not allow the flow to adjust rapidly enough to the motion of the body, while values that are too high may produce artificial stiffness. Moreover, for body motions that are not completely divergence-free a large pressure buildup is observed (see vande Voorde *et al.* (2004) for a case of lobe pumps). A major improvement was put forward by Mohd-Yusof (1997), who proposed to evaluate first the usual right-hand side for the flow equations at immersed points (or cells), and then add a force such that the velocity at the next timestep would satisfy the kinematic boundary condition $\mathbf{v} = \mathbf{w}_b$. Writing the spatially discretized form of the momentum equations at each point (or cell) i as

$$\mathbf{M}\frac{\Delta \mathbf{v}_i}{\Delta t} = \mathbf{r}_i + \mathbf{f}_i, \tag{18.3}$$

where $\mathbf{M}$, $\mathbf{v}$, $\mathbf{r}_i$ and $\mathbf{f}_i$ denote the mass matrix, nodal values of the velocity, right-hand side vector due to the momentum equations (advection, viscous terms, Boussinesque and gravity forces) and body force, respectively, $\mathbf{f}^i$ is obtained as

$$\mathbf{f}^i = \mathbf{M}\frac{\mathbf{w}_i^{n+1} - \mathbf{v}_i^n}{\Delta t} - \mathbf{r}^i. \tag{18.4}$$

Here $\mathbf{w}_i$ denotes the velocity of the immersed body at the location of point (or cell) i, and n the timestep. For explicit timestepping schemes, this force function in effect imposes the (required) velocity of the immersed body at the new timestep, implying that it can also be interpreted as a *kinematic* boundary condition treatment. Schemes of this kind have been used repeatedly (and very successfully) in conjunction with fractional step/projection methods for incompressible flow (Mohd-Yusof (1997), Angot *et al.* (1999), Fadlun *et al.* (2000), Kim *et al.* (2001), Löhner *et al.* (2004a,b), Balaras (2004), Tyagi and Acharya (2005), Mittal and Iaccarino (2005), Yang and Balaras (2006)). In this case, while the kinematic boundary condition $\mathbf{v}^{n+1} = \mathbf{w}^{n+1}$ is enforced strictly by (18.3) in the advective-diffusive prediction step, during the pressure correction step the condition is relaxed, offering the possibility of imposing the kinematic boundary conditions in a 'soft' way.

The imposition of a force given by (18.4) for all interior points will yield a first-order scheme for velocities (uncertainty of $O(h)$ in boundary location). This low-order boundary condition may be improved by extrapolating the velocity from the surface with field information to the layer of points surrounding the immersed body. The location where the flow velocity is equal to the surface velocity is the surface itself, and not the closest boundary point (Ye *et al.* (1999), Balaras (2004)). As shown in Figure 18.2, for each boundary point the closest point on the CSD face is found. Then, two (three) neighbouring field (i.e. non-boundary) points are found and a triangular (tetrahedral) element that contains the boundary point is formed. The velocity imposed at the field point is then obtained by interpolation. In this way, the boundary velocity 'lags' the field velocities by one iteration. This lagging of the velocities by one iteration can easily be implemented in any iterative solver.

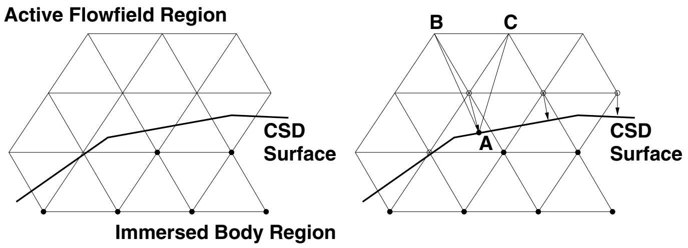


Figure 18.2. Extrapolation of velocity for immersed bodies

For cases where the bodies are not rigid, and all that is given is the embedded surface triangulation and movement, the force-terms added take the general form:

$$\mathbf{f} = \int_\Gamma \mathbf{F}\delta(\mathbf{x} - \mathbf{X}_\Gamma)\, d\Gamma, \tag{18.5}$$

where Γ denotes the location of the embedded surface, $\mathbf{X}_\Gamma$ the nearest embedded surface point to point $\mathbf{x}$ and $\mathbf{F}$ is the force. In theory, $\mathbf{F}$ should be applied to the fluid using a Dirac delta function δ in order to obtain a sharp interface. In most cases the influence of this delta-function is smeared over several grid points, giving rise to different methods (Glowinski *et al.* (1994), Bertrand *et al.* (1997), Patankar *et al.* (2000), Glowinski *et al.* (2000), Baiijens (2001)). If instead of a surface we are given the volume of the immersed body, then the penalization force may be applied at each point of the flow mesh that falls into the body.

While simple to program and employ, the force-based enforcement is particularly useful if the 'body thickness' covers several CFD mesh elements. This is because the pressures obtained are continuous (and computed!) across the embedded surface/immersed body. This implies that for thin embedded surfaces such as shells, where the pressure is different on both sides, this method will not yield satisfactory results (Löhner *et al.* (2007b)).

18.1.1. IMPLEMENTATION DETAILS

The search operations required for the imposition of kinetic boundary conditions are as follows:

- given a set of CSD faces (triangulation): find the edges of the CFD mesh cut by CSD faces (and possibly one to two layers of neighbours);
- given a set of CSD volumes (tetrahedrization): find the points of the CFD mesh that fall into a CSD tetrahedron (and possibly one to two layers of neighbours).

The first task is dealt with extensively below (see section 18.2.2(b)). Let us consider the second task in more detail. The problem can be solved in a number of ways.

(a) *Loop over the immersed body elements*

- Initialization:
  - store all CFD mesh points in a bin, octree, or any other similar data structure (see Chapter 2);
- Loop over the immersed body elements:
  - determine the bounding box of the element;
  - find all points in the bounding box;
  - detailed analysis to determine the shape function values.

(b) *Loop over the CFD mesh points*

- Initialization:
  - store all immersed body elements in a bin, modified octree, or any other similar data structure;
- Loop over the CFD mesh points:
  - obtain the elements in the vicinity of the point;
  - detailed analysis to determine the shape function values.

In both cases, if the immersed body only covers a small portion of the CFD domain, one can reduce the list of points stored or points checked via the bounding box of all immersed body points. Both approaches are easily parallelized on shared-memory machines.

18.2. Kinematic treatment of embedded surfaces

Embedded surfaces may alternatively be treated by applying ***kinematic boundary conditions*** at the nodes close to the embedded surface. Depending on the required order of accuracy and simplicity, a first- or second-order (higher-order) scheme may be chosen to apply the kinematic boundary conditions. Figures 18.3(a) and (b) illustrate the basic difference between these approaches for edge-based solvers. Note that in both cases the treatment of infinitely thin surfaces with fluid on both sides (e.g. fluid–structure interaction simulations with shells) is straightforward.

A first-order scheme can be achieved by:

- eliminating the edges crossing the embedded surface;
- forming boundary coefficients to achieve flux balance;
- applying boundary conditions for the endpoints of the crossed edges based on the normals of the embedded surface.

A second-order scheme can be achieved by:

- duplicating the edges crossing the embedded surface;
- duplicating the endpoints of crossed edges;
- applying boundary conditions for the endpoints of the crossed edges based on the normals of the embedded surface.

In either case CFD edges crossed by CSD faces are modified/duplicated. Given that an edge/face crossing is essentially the same in two and three dimensions, these schemes are rather general.

The following sections describe in more detail each one of the steps required, as well as near-optimal techniques to realize them.

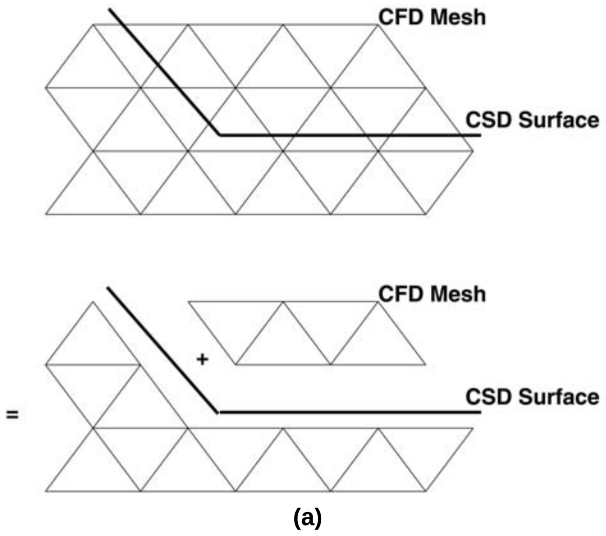
18.2.1. FIRST-ORDER TREATMENT

The first-order scheme is the simplest to implement. Given the CSD triangulation and the CFD mesh, the CFD edges cut by CSD faces are found and deactivated. Considering an arbitrary field point i, the time advancement of the unknowns $\mathbf{u}^i$ for an explicit edge-based time integration scheme is given by

$$M^i \Delta \mathbf{u}^i = \Delta t \sum_{ij_\Omega} C^{ij}(F_i + F_j). \tag{18.6}$$

Here C, F and M denote, respectively, the edge coefficients, fluxes and mass matrix. For any edge ij crossed by a CSD face, the coefficients C^{ij} are set to zero. This implies that for a uniform state $\mathbf{u} =$ constant the balance of fluxes for interior points with cut edges will not vanish. This is remedied by defining a new boundary point to impose total/normal velocities, as well as adding a ‘boundary contribution’, resulting in

$$M^i \Delta \mathbf{u}^i = \Delta t \left[\sum_{ij_\Omega} C^{ij}(F_i + F_j) + C^i_\Gamma F_i \right]. \tag{18.7}$$



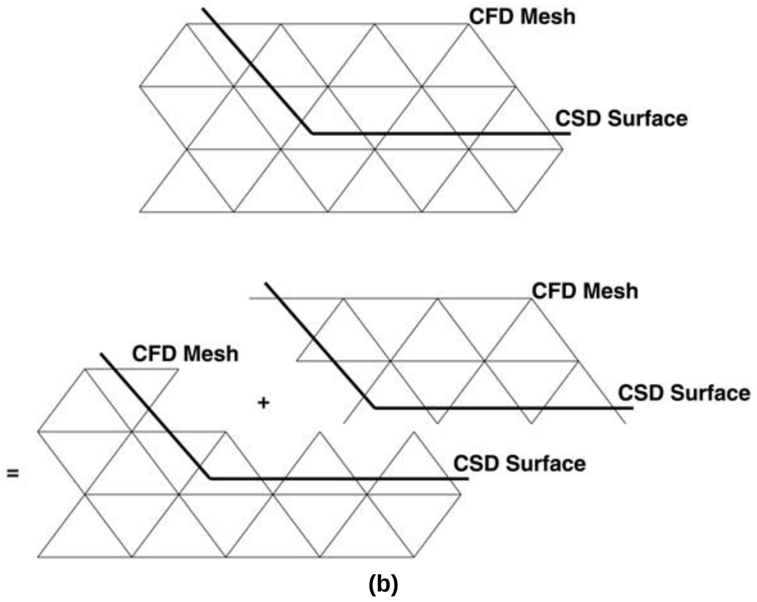


Figure 18.3. (a) First-order treatment of embedded surfaces; (b) second-order treatment of embedded surfaces

The ‘boundary contribution’ is the numerical equivalent of the boundary integral that would appear if a boundary fitted mesh had been placed instead. The point coefficients C^i_Γ are obtained from the condition that $\Delta\mathbf{u} = 0$ for $\mathbf{u} = \text{constant}$. Given that gradients (e.g. for limiting) are also constructed using a loop of the form given by (18.6) as

$$M^i \mathbf{g}^i = \sum_{ij_\Omega} \mathbf{C}^{ij}(u_i + u_j), \tag{18.8}$$

it would be desirable to build the C^i_Γ coefficients in such a way that the constant gradient of a linear function u can be obtained exactly. However, this is not possible, as the number of coefficients is too small. Therefore, the gradients at the boundary are either set to zero or extrapolated from the interior of the domain.

The mass matrix M^i of points surrounded by cut edges must be modified to reflect the reduced volume due to cut elements. The simplest possible modification of M^i is given by the so-called ‘cut edge fraction’ method.

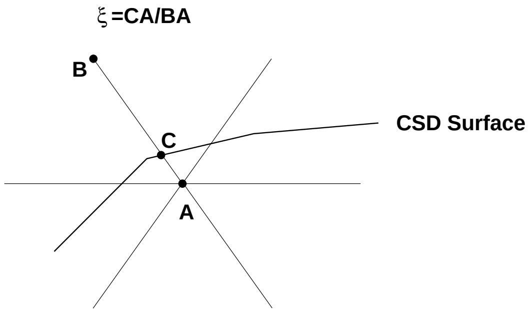


Figure 18.4. Cut edge fraction

In a pass over the edges, the smallest ‘cut edge fraction’ ξ for all the edges surrounding a point is found (see Figure 18.4). The modified mass matrix is then given by

$$M^i_* = \frac{1+\xi_{\min}}{2} M^i. \tag{18.9}$$

Note that the value of the modified mass matrix can never fall below half of its original value, implying that timestep sizes will always be acceptable. For edges that are along embedded boundaries (see Figure 18.5), the original edge coefficient must be modified to reflect the loss of volume resulting from the presence of the boundary. In principle, a complex quadrature may be used to recompute the edge coefficients. A very simple approach to obtain an approximation to these values is again via the ‘cut edge fraction’ technique. Given the ‘cut edge fraction’ of the endpoints, the C^{ij} are modified as follows:

$$C^{ij}_* = \frac{1+\max(\xi_i, \xi_j)}{2} C^{ij}. \tag{18.10}$$

18.2.1.1. Boundary conditions

For the new boundary points belonging to cut edges the proper PDE boundary conditions are required. In the case of flow solvers, these are either an imposed velocity or an imposed normal velocity. For limiting and higher-order schemes, one may also have to impose boundary conditions on the gradients. The required surface normal and boundary velocity are obtained directly from the closest CSD face to each of the new boundary points.

These low-order boundary conditions may be improved by extrapolating the velocity from the surface with field information. The location where the flow velocity is equal to

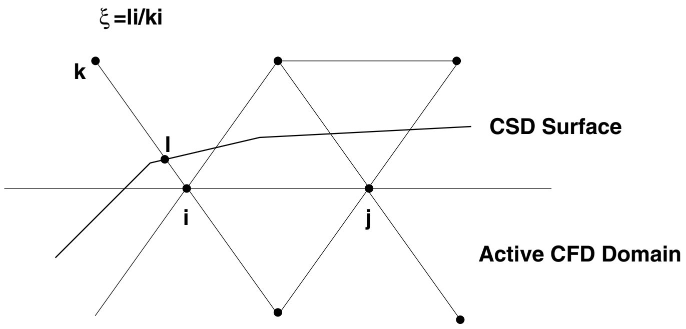


Figure 18.5. Modification of boundary edge coefficients

the surface velocity is the surface itself, and not the closest boundary point. As shown in Figure 18.6, for each boundary point the closest point on the CSD face is found. Then, two (three) neighbouring field (i.e. non-boundary) points are found and a triangular (tetrahedral) element that contains the boundary point is formed. The velocity imposed at the field point is then obtained by interpolation. In this way, the boundary velocity 'lags' the field velocities by one iteration (for iterative implicit schemes) or timestep (for explicit schemes).

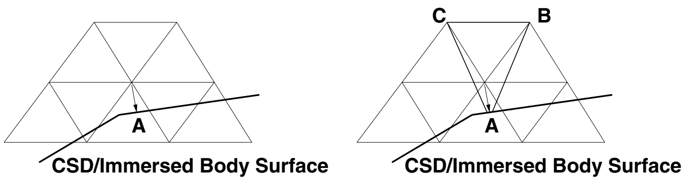


Figure 18.6. Extrapolation of velocity

The normal gradients at the boundary points can be improved by considering the 'most aligned' field (i.e., non-boundary) point to the line formed by the boundary point and the closest point on the CSD face (see Figure 18.7).

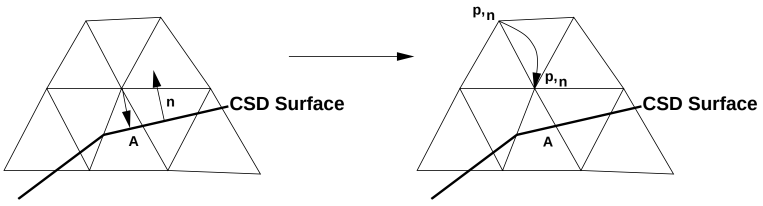


Figure 18.7. Extrapolation of normal pressure gradient

18.2.2. HIGHER-ORDER TREATMENT

As stated before, a higher-order treatment of embedded surfaces may be achieved by using ghost points or mirrored points to compute the contribution of the crossed edges to the overall solution. This approach presents the advantage of not requiring the modification of the mass matrix as all edges (even the crossed ones) are taken into consideration. Neither does it require an extensive modification of the various solvers. On the other hand, it requires more memory due to duplication of crossed edges and points, as well as (scalar) CPU time for renumbering/reordering arrays. For moving body problems, in particular, this may represent a considerable CPU burden.

18.2.2.1. Boundary conditions

By duplicating the edges, the points are treated in the same way as in the original (non-embedded) case. The boundary conditions are imposed indirectly by mirroring and interpolating the unknowns as required. Figure 18.8 depicts the contribution due to the edges surrounding point i. A CSD boundary crosses the CFD domain. In this particular situation point j, which lies on the opposite side of the CSD face, will have to use the flow values of its mirror image j' based on the crossed CSD face.

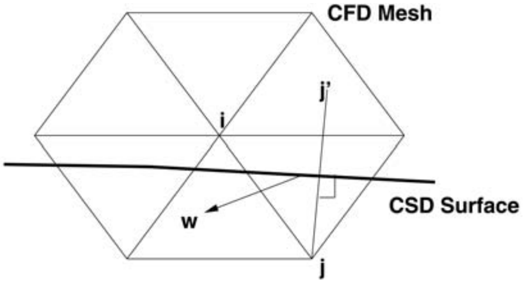


Figure 18.8. Higher-order boundary conditions

The flow values of the mirrored point are then interpolated from the element the point resides in using the following formulation for inviscid flows:

$$\rho_m = \rho_i, \quad p_m = p_i, \quad \mathbf{v}_m = \mathbf{v}_i - 2[(\mathbf{v}_i - \mathbf{w}_{\text{csd}}) \cdot \mathbf{n}]\mathbf{n}, \tag{18.11}$$

where $\mathbf{w}_{\text{csd}}$ is the velocity of the crossed CSD face, ρ the density, $\mathbf{v}$ the flow velocity, p the pressure and $\mathbf{n}$ the unit surface normal of the face. Proper handling of the interpolation for degenerate cases is also required, as the element used for the interpolation might either be crossed (Figure 18.9(a)) or not exist (Figure 18.9(b)).

A more accurate formulation of the mirrored pressure and density can also be used taking into account the local radius of curvature of the CSD wetted surface:

$$p_m = p_i - \rho_i \frac{[\mathbf{v}_i - (\mathbf{v}_i - \mathbf{w}_{\text{csd}}) \cdot \mathbf{n}]^2}{R_i} \Delta, \quad \rho_m = \rho_i \left(\frac{p_m}{p_i} \right)^{1/\gamma}, \tag{18.12}$$

where R_i is the radius of curvature and Δ the distance between the point and its mirror image. This second formulation is more complex and requires the computation of the two radii (3D) of curvature at each CSD point (Dadone and Grossman (2002)). The radius of curvature plays an important role for large elements but this influence can be diminished by the use of automatic h-refinement.

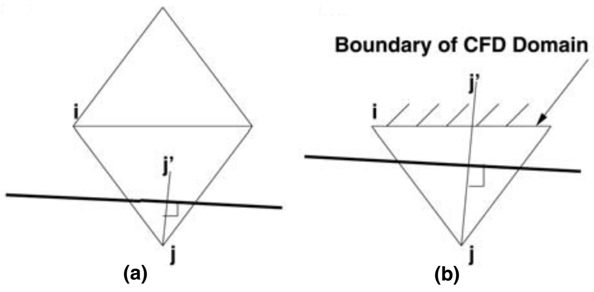


Figure 18.9. Problem cases

For problematic cases such as the one shown in Figure 18.9, the interpolation will be such that the point at which the information is interpolated may not be located at the same normal distance from the wall as the point where information is required.

With the notation of Figure 18.10, and assuming a linear interpolation of the velocities, the velocity values for the viscous (i.e. no-slip) case are interpolated as

$$\mathbf{w} = (1-\xi_w)\mathbf{v}_c + \xi_w \mathbf{v}_i; \quad \xi_w = \frac{h_o}{h_o + h_i}, \tag{18.13}$$

i.e.

$$\mathbf{v}_c = \frac{1}{1-\xi_w}\mathbf{w} - \frac{\xi_w}{1-\xi_w}\mathbf{v}_i. \tag{18.14}$$

Here **w** is the average velocity of the crossed CSD face, $\mathbf{v}_i$ the interpolated flow velocity and the distance factor $\xi_w \leq 0.5$.

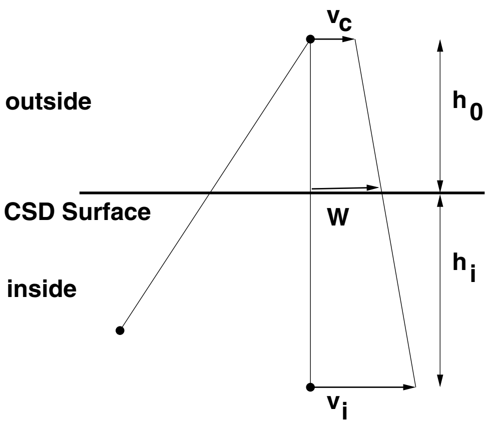


Figure 18.10. Navier–Stokes boundary condition

18.2.3. DETERMINATION OF CROSSED EDGES

Given the CSD triangulation and the CFD mesh, the first step is to find the CFD edges cut by CSD faces. This is performed by building a fast spatial search data structure, such as an octree or a bin for the CSD faces (see Figure 18.11). Without loss of generality, let us assume a bin for the CSD faces. Then a (parallel) loop is performed over the edges. For each edge, the bounding box of the edge is built.

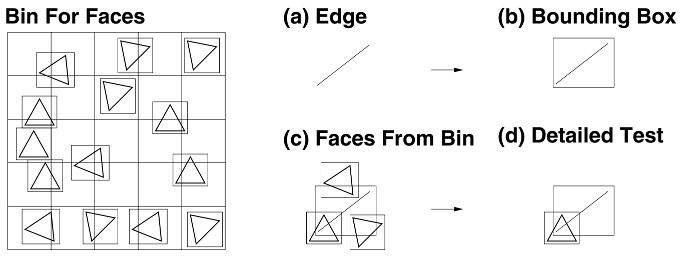


Figure 18.11. Bin storage of faces and search

From the bin, all the faces in the region of the bounding box are found. This is followed by an in-depth test to determine which faces cross the given edge. The crossing face closest to each of the edge end-nodes is stored. This allows to resolve cases of thin gaps or cusps. Once the faces crossing edges are found, the closest face to the endpoints of crossed edges is also stored. This information is required to apply boundary conditions for the points close to the embedded surface. For cases where the embedded surfaces only cut a small portion of the CFD edges, a considerable speedup may be realized by removing from the list of edges tested all those that fall outside the global bounding box of the CSD faces. The resulting list of edges to be tested in depth may be reduced further by removing all edges whose bounding boxes do not fall into an octree or bin filled with faces covering that spatial region. One typically finds that, using these two filters, the list of edges to be tested in detail has been reduced by an order of magnitude.

For transient problems, the procedure described above can be improved considerably. The key assumption is that the CSD triangulation will not move over more than one to two elements during a timestep. If the topology of the CSD triangulation has not changed, the crossed-edge information from the previous timestep can be re-checked. The points of edges no longer crossed by a face crossing them in the previous timestep are marked, and the neighbouring edges are checked for crossing. If the topology of the CSD triangulation has changed, the crossed-edge information from the previous timestep is no longer valid. However, the points close to cut edges in the previous timestep can be used to mark one or two layers of edges. Only these edges are then re-checked for crossing.

18.3. Deactivation of interior regions

For highly distorted CSD surfaces, or for CSD surfaces with thin re-entrant corners, all edges surrounding a given point may be crossed by CSD faces (see Figure 18.12). The best way to treat such points is to simply deactivate them.

This deactivation concept can be extended further in order to avoid unnecessary work for regions inside solid objects. Several approaches have been pursued in this direction: seed points and automatic deactivation.

(a) *Seed points*. In this case, the user specifies a point inside an object. The closest CFD field point to this so-called seed point is then obtained. Starting from this point, additional

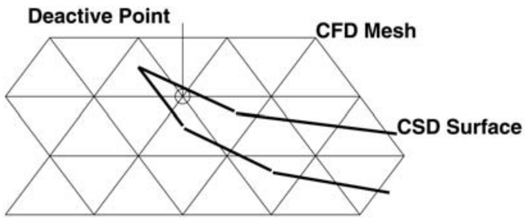


Figure 18.12. Deactive point

points are added using an advancing-front (nearest-neighbour layer) algorithm, and flagged as inactive. The procedure stops once points that are attached to crossed edges have been reached.

(b) *Automatic seed points*. For external aerodynamics problems, one can define seed points on the farfield boundaries. At the beginning, all points are marked as deactive. Starting from the external boundaries, points are activated using an advancing front technique into the domain. The procedure only activates neighbour points if at least one of the edges is active and attached to an active point. All unmarked points and edges are then deactivated.

(c) *Automatic deactivation*. For complex geometries with moving surfaces, the manual specification of seed points becomes impractical. An automatic way of determining which regions correspond to the flowfield region one is trying to compute and which regions correspond to solid objects immersed in it is then required. The algorithm starts from the edges crossed by embedded surfaces. For the endpoints of these edges an in/outside determination is attempted. This is non-trivial, particularly for thin or folded surfaces (Figure 18.13). A more reliable way

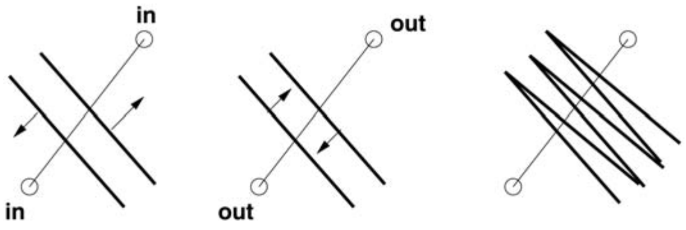


Figure 18.13. Edges with multiple crossing faces

to determine whether a point is in/outside the flowfield is obtained by storing, for the crossed edges, the faces closest to the endpoints of the edge. Once this in/outside determination has been done for the endpoints of crossed edges, the remaining points are marked using an advancing front algorithm. It is important to remark that in this case both the inside (active) and outside (deactive) points are marked at the same time. In the case of a conflict, preference is always given to mark the points as inside the flow domain (active). Once the points have been marked as active/inactive, the element and edge groups required to avoid memory contention (i.e. to allow vectorization) are inspected in turn. As with space-marching (see Chapter 16, as well as Nakahashi and Saitoh (1996), Löhner (1998), Morino and Nakahashi (1999)) the idea is to move the active/inactive `if`-tests to the element/edge group level in order to simplify and speed up the core flow solver.

18.4. Extrapolation of the solution

For problems with moving boundaries, mesh points can switch from one side of a surface to another or belong/no longer belong to an immersed body (see Figure 18.14). For these cases, the solution must be extrapolated from the proper state. The conditions that have to be met for extrapolation are as follows:

- the edge was crossed at the previous timestep and is no longer crossed;
- the edge has one field point (the point donating unknowns) and one boundary point (the point receiving unknowns); and
- the CSD face associated with the boundary point is aligned with the edge.

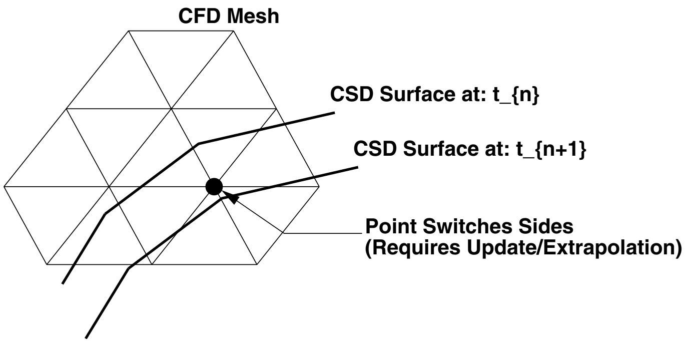


Figure 18.14. Extrapolation of solution

For incompressible flow problems the simple extrapolation of the solution from one point to another (or even a more sophisticated extrapolation using multiple neighbours) will not lead to a divergence-free velocity field. Therefore, it may be necessary to conduct a local ‘divergence cleanup’ for such cases.

18.5. Adaptive mesh refinement

Adaptive mesh refinement is very often used to reduce CPU and memory requirements without compromising the accuracy of the numerical solution. For transient problems with moving discontinuities, adaptive mesh refinement has been shown to be an essential ingredient of production codes (Baum *et al.* (1999), Löhner *et al.* (1999a,c)). For embedded CSD triangulations, the mesh can be refined automatically close to the surfaces (Aftosmis *et al.* (2000), Löhner *et al.* (2004b)). One can define a number of refinement criteria, of which the following have proven to be the most useful:

- refine the elements with edges cut by CSD faces to a certain size/level;
- refine the elements so that the curvature given by the CSD faces can be resolved (e.g. 10 elements per 90° bend);

- refine the elements close to embedded CSD corners with angles up to 50° to a certain size/level;
- refine the elements close to embedded CSD ridges with angles up to 15° to a certain size/level.

The combination of adaptive refinement and embedded/immersed grid techniques has allowed the complete automation of certain application areas, such as external aerodynamics. Starting from a triangulation (e.g. an STL data set obtained from CAD for a design or a triangulation obtained from a scanner for reverse engineering), a suitable box is automatically generated and filled with optimal space-filling tetrahedra. This original mesh is then adaptively refined according to the criteria listed above. The desired physical and boundary conditions for the fluid are read in, and the solution is obtained. In some cases, further mesh adaptation based on the physics of the problem (shocks, contact discontinuities, shear layers, etc.) may be required, but this can also be automated to a large extent for class-specific problems. Note that the only user input consists of flow conditions. The many hours required to obtain a watertight, consistent surface description have thus been eliminated by the use of adaptive, embedded flow solvers.

18.6. Load/flux transfer

For fluid–structure interaction problems, the forces exerted by the fluid on the embedded surfaces or immersed bodies need to be evaluated. For *immersed bodies*, this information is given by the sum of all forces, i.e. by (18.4). For *embedded surfaces*, the information is obtained by computing first the stresses (pressure, shear stresses) in the fluid domain, and then interpolating this information to the embedded surface triangles. In principle, the integration of forces can be done with an arbitrary number of Gauss points per embedded surface triangle. In practice, one Gauss point is used most of the time. The task is then to interpolate the stresses to the Gauss points on the faces of the embedded surface. Given that the information of crossed edges is available, the immediate impulse would be to use this information to obtain the required information. However, this is not the best way to proceed, as:

- the closest (endpoint of crossed edge) point corresponds to a low-order solution and/or stress, i.e. it may be better to interpolate from a nearby field point;
- as can be seen from Figure 18.15, a face may have multiple (F1) or no (F2) crossing edges, i.e. there will be a need to construct extra information in any case.

For each Gauss point required, the closest interpolating points are obtained as follows.

- Obtain a search region to find close points; this is typically of the size of the current face the Gauss point belongs to, and is enlarged or reduced depending on the number of close points found.
- Obtain the close faces of the current surface face.
- Remove from the list of close points those that would cross close faces that are visible from the current face, and that can in turn see the current face (see Figure 18.16).

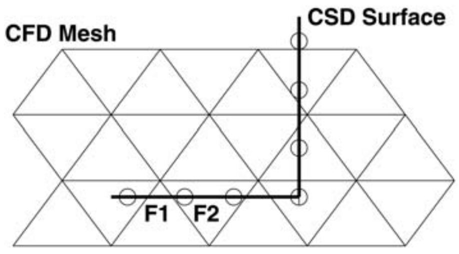


Figure 18.15. Transfer of stresses/fluxes

- Order the close points according to proximity and boundary/field point criteria.
- Retain the best n_p close points from the ordered list.

The close points and faces are obtained using octrees for the points and a modified octree or bins for the faces. Experience indicates that for many fluid–structure interaction cases this step can be more time-consuming than finding the crossed edges and modifying boundary conditions and arrays, particularly if the characteristic size of the embedded CSD faces is smaller than the characteristic size of the surrounding tetrahedra of the CFD mesh, and the embedded CSD faces are close to each other and/or folded.

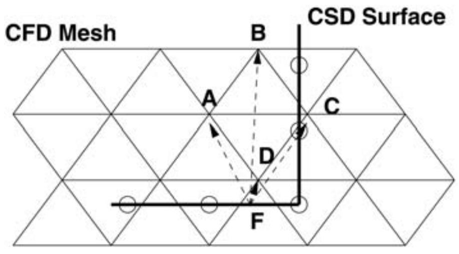


Figure 18.16. Transfer of stresses/fluxes

18.7. Treatment of gaps or cracks

The presence of 'thin regions' or gaps in the surface definition, or the appearance of cracks due to fluid–structure interactions has been an outstanding issue for a number of years. For body-fitted grids (Figure 18.17(a)), a gap or crack is immediately associated with minuscule grid sizes, small timesteps and increased CPU costs.

For embedded grids (Figure 18.17(b)), the gap or crack may not be seen. A simple solution is to allow some flow through the gap or crack without compromising the timestep. The key idea is to change the geometrical coefficients of crossed edges in the presence of gaps. Instead of setting these coefficients to zero, they are reduced by a factor that is proportional to the size of the gap δ to the average element size h in the region:

$$C_k^{ij} = \eta C_{0k}^{ij}; \quad \eta = \delta/h. \tag{18.15}$$

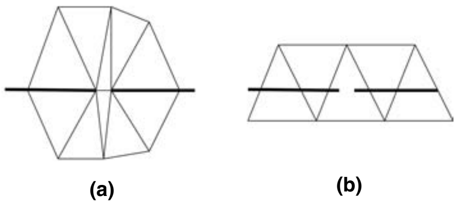


Figure 18.17. Treatment of gaps: (a) body fitted; (b) embedded

Gaps are detected by considering the edges of elements with multiple crossed edges. If the faces crossing these edges are different, a test is performed to see whether one face can be reached by the other via a near-neighbour search. If this search is successful, the CSD surface is considered watertight. If the search is not successful, the gap size δ is determined and the edges are marked for modification.

18.8. Direct link to particles

One of the most promising ways to treat discontinua is via so-called discrete element methods (DEMs) or discrete particle methods (DPMs). A considerable amount of work has been devoted to this area in the last two decades, and these techniques are being used for the prediction of soil, masonry, concrete and particulates (Cook and Jensen (2002)). The filling of space with objects of arbitrary shape has also reached the maturity of advanced unstructured grid generators (Löhner and Oñate (2004b Chapter 2)), opening the way for widespread use with arbitrary geometries. Adaptive embedded grid techniques can be linked to DPMs in a very natural way. The discrete particle is represented as a sphere. Discrete elements, such as polyhedra, may be represented as an agglomeration of spheres. The host of the centroid of each discrete particle is updated every timestep and is assumed as given. All points of host elements are marked for additional boundary conditions. The closest particle to each of these points is used as a marker. Starting from these points, all additional points covered by particles are marked (see Figure 18.18).

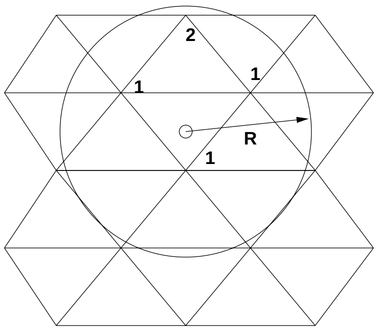


Figure 18.18. Link to discrete particle method

All edges touching any of the marked points are subsequently marked as crossed. From this point onwards, the procedure reverts back to the usual embedded mesh or immersed body techniques. The velocity of particles is either imposed at the endpoints of crossed edges (embedded) or for all points inside and surrounding the particles (immersed).

18.9. Examples

Adaptive embedded and immersed unstructured grid techniques have been used extensively for a number of years now. The aim of this section is to highlight the considerable range of applicability of these methods, and show their limitations as well as the combination of body-fitted and embedded/immersed techniques. We start with compressible, inviscid flow, where we consider the classic Sod shock tube problem, a shuttle ascend configuration and two fluid–structure interaction problems. We then consider incompressible, viscous flow, where we show the performance of the different options in detail for a sphere. The contaminant transport calculation for a city is then included to show a case where obtaining a body-fitted, watertight geometry is nearly impossible. Finally, we show results obtained for complex endovascular devices in aneurysms, as well as the external flow past a car.

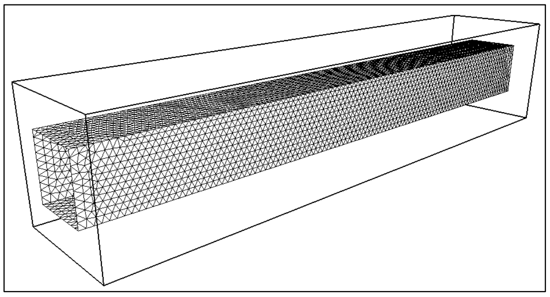
18.9.1. SOD SHOCK TUBE

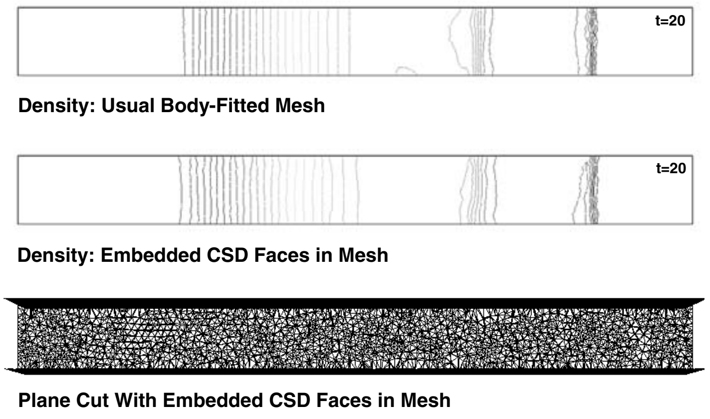
The first case considered is the classic Sod (1978) shock tube problem ($\rho_1 = p_1 = 1.0$, $\rho_2 = p_2 = 0.1$) for a ‘clean wall’, body-fitted mesh and an equivalent embedded CSD mesh. The flow is treated as compressible and inviscid, with no-penetration (slip) boundary conditions for the velocities at the walls.

The embedded geometry can be discerned from Figure 18.19(a). Figure 18.19(b) shows the results for the two techniques. Although the embedded technique is rather primitive, the results are surprisingly good. The main difference is slightly more noise in the contact discontinuity region, which may be expected, as this is a linear discontinuity. The long-term effects on the solution for the different treatments of boundary points can be seen in Figure 18.19(c), which shows the pressure time history for a point located on the high-pressure side (left of the membrane). Both ends of the shock tube are assumed closed. One can see the different reflections. In particular, the curves for the boundary-fitted approach and the second-order (ghost-point) embedded approach are almost identical, whereas the first-order embedded approach exhibits some damping.

18.9.2. SHUTTLE ASCEND CONFIGURATION

The second example considered is the Space Shuttle ascend configuration shown in Figure 18.20(a). The external flow is at $Ma = 2$ and the angle of attack $\alpha = 5°$. As before, the flow is treated as compressible and inviscid, with no-penetration (slip) boundary conditions for the velocities at the walls. The surface definition consisted of approximately 161 000 triangular faces. The base CFD mesh had approximately 1.1 million tetrahedra. For the geometry, a minimum of three levels of refinement were specified. Additionally, curvature-based refinement was allowed up to five levels. This yielded a mesh of approximately 16.9 million tetrahedra. The grid obtained in this way, as well as the corresponding solution, are shown in Figures 18.20(b) and (c). Note that all geometrical details have been properly resolved. The mesh was subsequently refined based on a modified interpolation theory error indicator (Löhner (1987), Löhner and Baum (1992)) of the density, up to approximately 28 million tetrahedra. This physics-based mesh refinement is evident in Figures 18.20(d)–(f).




t=20
Density: Usual Body-Fitted Mesh
t=20
Density: Embedded CSD Faces in Mesh
Plane Cut With Embedded CSD Faces in Mesh
(b)

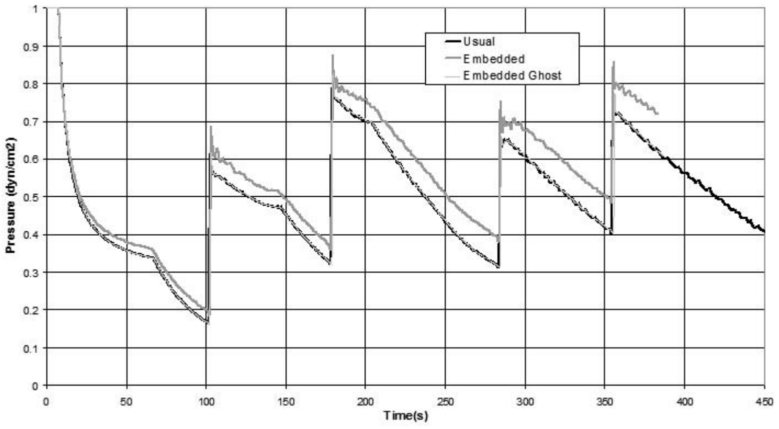


Figure 18.19. Shock tube problem: (a) embedded surface; (b) density contours; (c) pressure time history

18.9.3. BLAST INTERACTION WITH A GENERIC SHIP HULL

Figure 18.21 shows the interaction of an explosion with a generic ship hull. For this fully coupled CFD/CSD run, the structure was modelled with quadrilateral shell elements, the

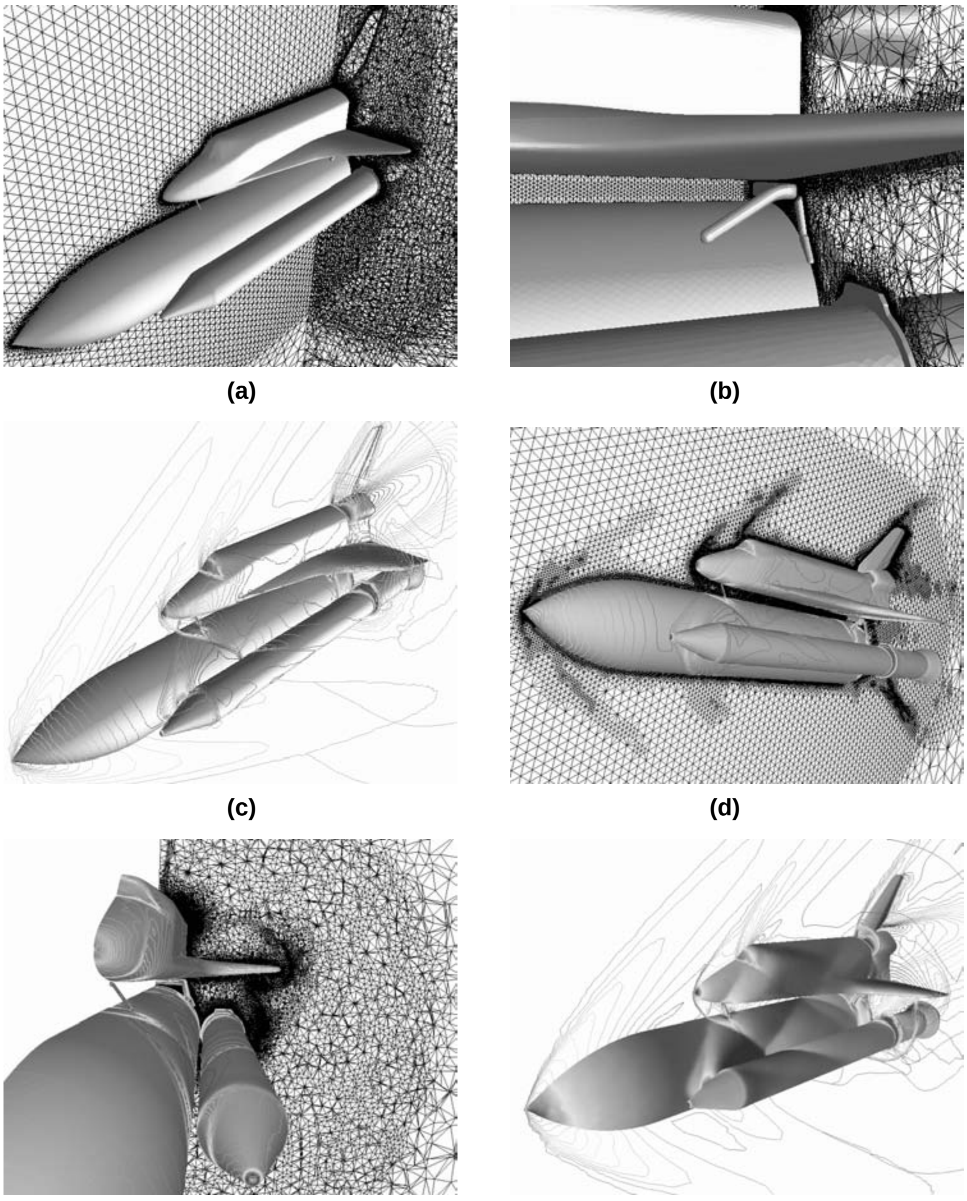


Figure 18.20. Shuttle: (a) general view and (b) detail; (c), (f) surface pressure and field Mach number; (d), (e) surface pressure and mesh (cut plane)

(inviscid) fluid was taken as a mixture of high explosive and air and mesh embedding was employed. The structural elements were assumed to fail once the average strain in an element exceeded 60%. As the shell elements failed, the fluid domain underwent topological changes.

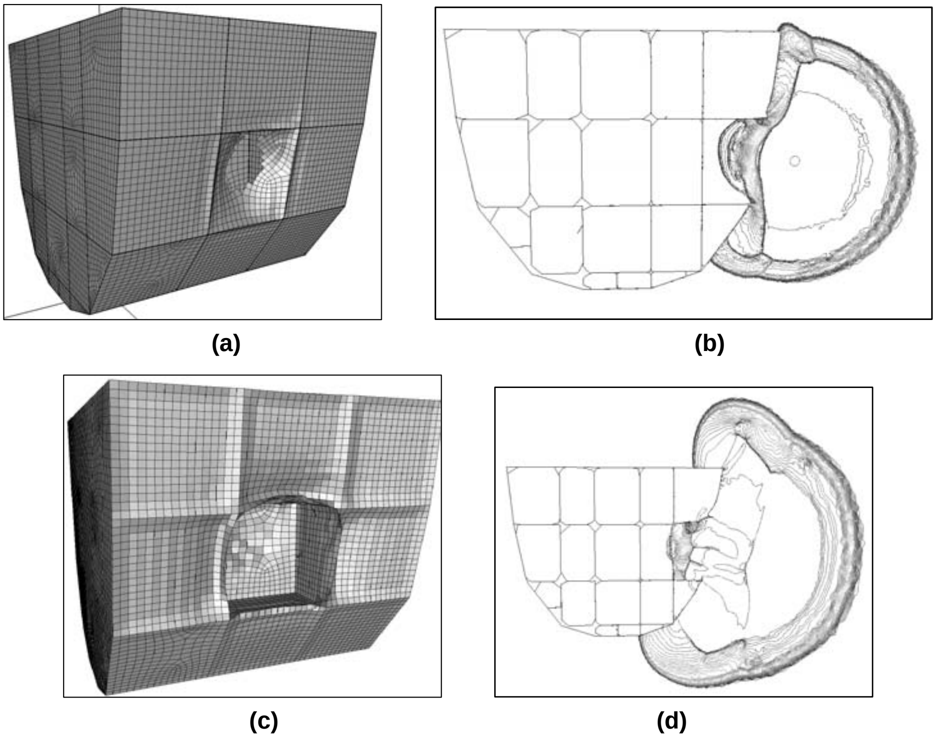


Figure 18.21. (a) Surface and (b) pressure in cut plane at 20 ms; (c) surface and (d) pressure in cut plane at 50 ms

Figures 18.21(a)–(d) show the structure as well as the pressure contours in a cut plane at two times during the run. The influence of bulkheads on surface velocity can clearly be discerned. Note also the failure of the structure, and the invasion of high pressure into the chamber. The distortion and inter-penetration of the structural elements is such that the traditional moving mesh approach (with topology reconstruction, remeshing, ALE formulation, remeshing, etc.) will invariably fail for this class of problems. In fact, it was this particular type of application that led to the development of embedded/immersed CSD techniques for unstructured grids.

18.9.4. GENERIC WEAPON FRAGMENTATION

Figure 18.22 shows a generic weapon fragmentation study. The CSD domain was modelled with approximately 66 000 hexagonal elements corresponding to 1555 fragments whose mass distribution matches statistically the mass distribution encountered in experiments. The structural elements were assumed to fail once the average strain in an element exceeded 60%. The high explosive was modelled with a Jones–Wilkins–Lee equation of state (Löhner *et al.* (1999c)). The CFD mesh was refined to three levels in the vicinity of the solid surface. Additionally, the mesh was refined based on the modified interpolation error indicator (Löhner (1987), Löhner and Baum (1992)) using the density as an indicator variable.

Adaptive refinement was invoked every five timesteps during the coupled CFD/CSD run. The CFD mesh started with 39 million tetrahedra, and ended with 72 million tetrahedra.

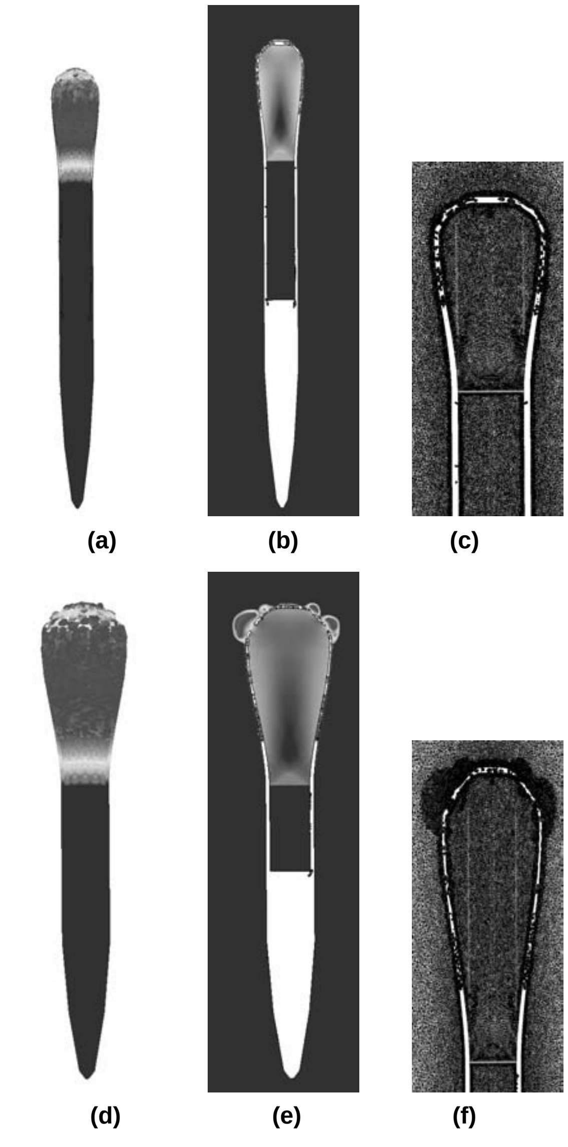
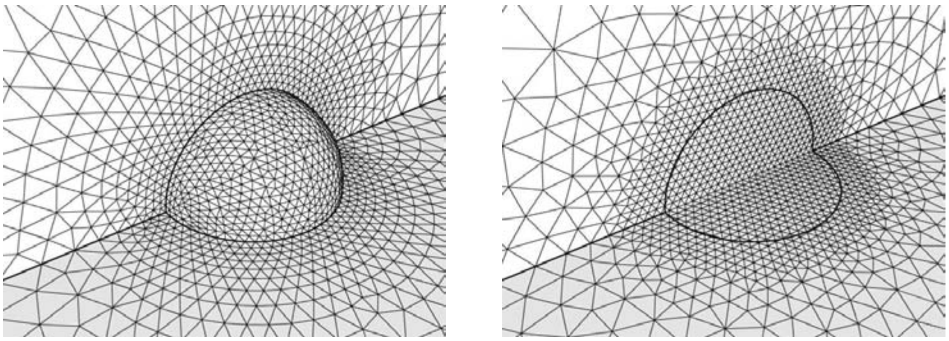


Figure 18.22. (a), (b), (c): CSD/flow velocity and pressure/mesh at 68 ms; (d), (e), (f): CSD/flow velocity and pressure/mesh at 102 ms

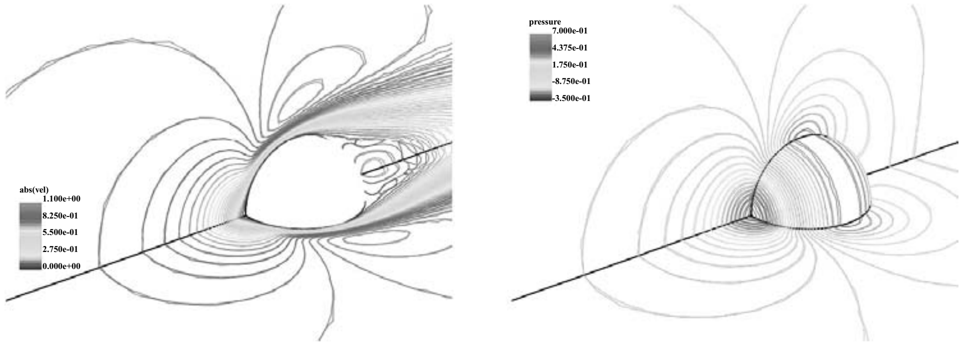
Figures 18.22(a)–(f) show the structure as well as the pressure contours in a cut plane at two times during the run. The detonation wave is clearly visible, as well as the thinning of the structural walls and the subsequent fragmentation.

18.9.5. FLOW PAST A SPHERE

This simple case is included here as it offers the possibility of an accurate comparison of the different techniques discussed in this chapter. The geometry considered is shown



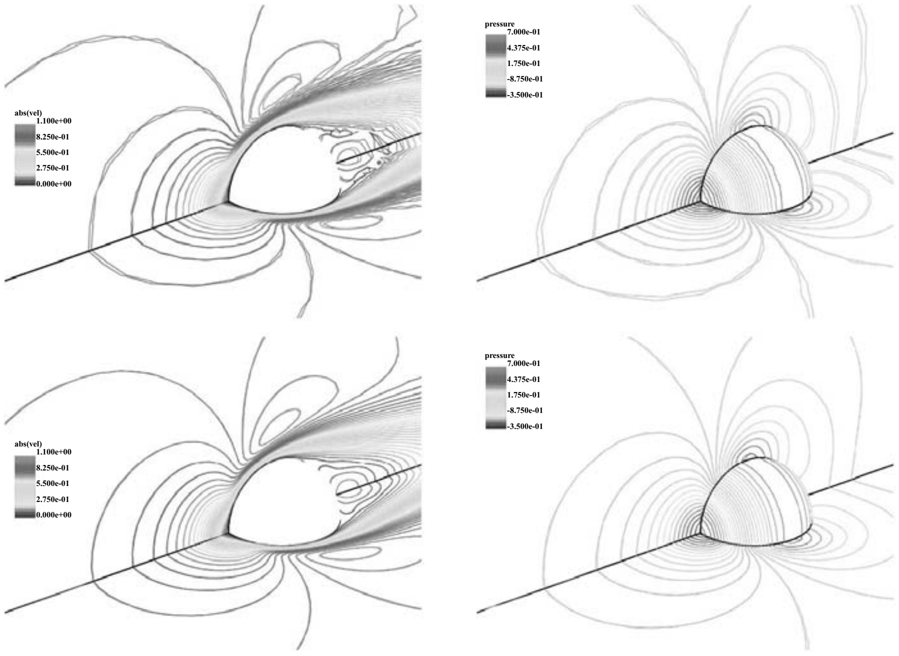
(a)



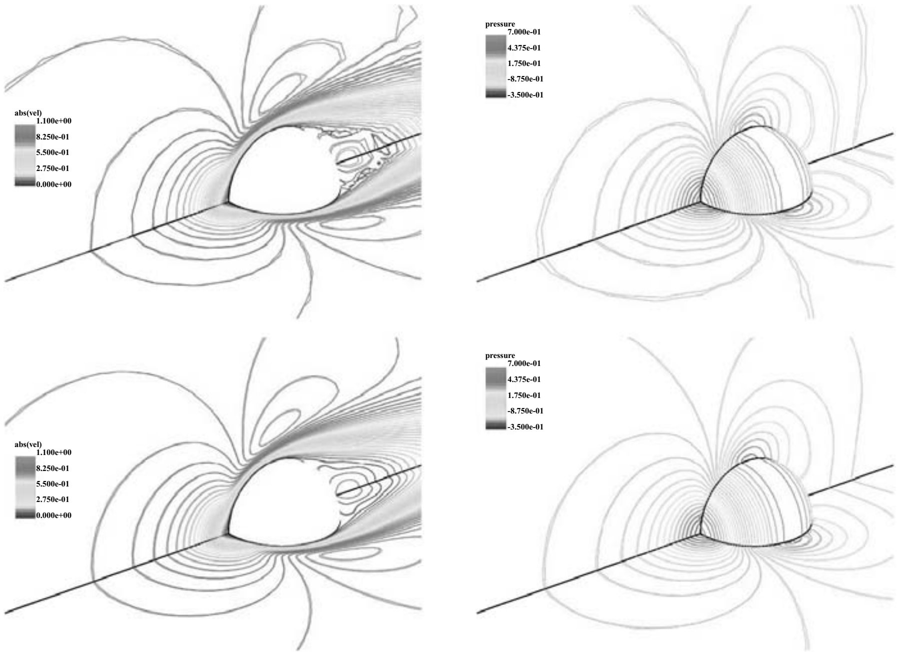
(b)

Figure 18.23. Sphere: (a) surface grids (body fitted, embedded/immersed); (b) coarse versus fine body-fitted (left, $|\mathbf{v}|$; right, p); (c) body-fitted versus embedded 1 (top, coarse; bottom, fine); (d) body-fitted versus embedded 2 (top, coarse; bottom, fine); (e) body-fitted versus immersed (top, coarse; bottom, fine); (f) velocity/pressure along the line $y, z = 0.0$ (top, coarse; bottom, fine)

in Figure 18.23. Due to symmetry considerations only a quarter of the sphere is treated. The physical parameters were set as $D = 1$, $\mathbf{v}_\infty = (1, 0, 0)$, $\rho = 1.0$, $\mu = 0.01$, yielding a Reynolds number of $Re = 100$. Two grids were considered: the first had an element size of approximately $h = 0.0450$ in the region of the sphere, while the corresponding size for the second was $h = 0.0225$. This led to grids with approximately 140 000 elements and 1.17 million elements, respectively. The coarse mesh surface grids for the body-fitted and embedded options are shown in Figure 18.23(a). It was implicitly assumed that the body-fitted results were more accurate and therefore these were considered as the 'gold standard'. Figure 18.23(b) shows the same 50 surface contour lines of the absolute value of the velocity, as well as the pressures, obtained for the body-fitted coarse and fine grids. Note that, although some differences are apparent, the results are quite close, indicating a grid-converged result on the fine mesh. The drag coefficients for the two body-fitted grids were given by $c_D = 1.07$ and $c_D = 1.08$, respectively, in good agreement with experimental results (Schlichting (1979)). Figures 18.23(c)–(e) show the same surface contour lines for the body-fitted and the different embedded/immersed options for the two grids. Note that the contours are very close, and in most cases almost identical. This is particularly noticeable for the second-order embedded

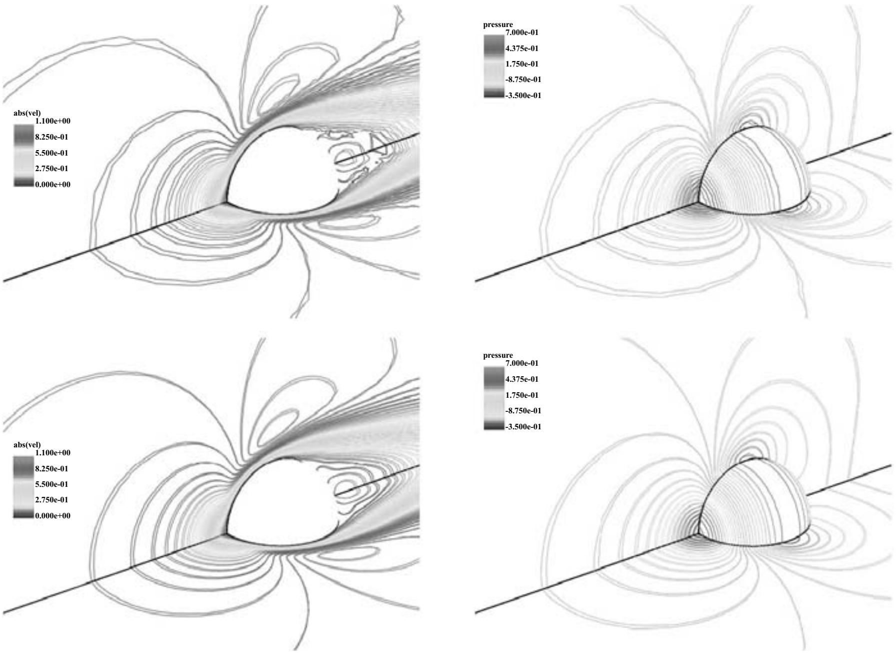


(c)

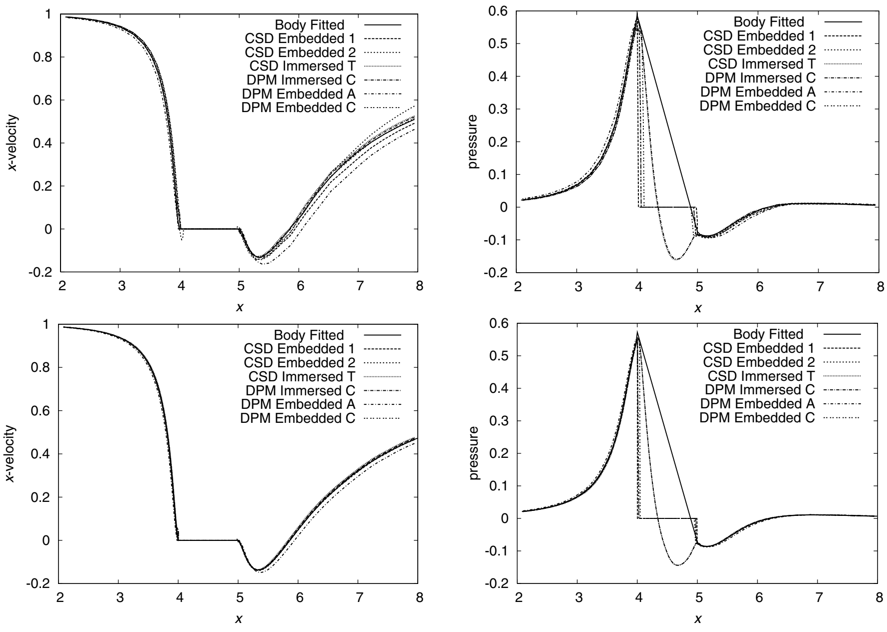


(d)

Figure 18.23. Continued



(e)



(f)

Figure 18.23. Continued

and the immersed body cases. Figure 18.23(f) depicts the x-velocity and pressure along the line $y, z = 0$ (i.e. the axis of symmetry). As seen before in the contour lines, the results are very close, indicating that even the first-order embedded scheme has converged. For more information, see Löhner *et al.* (2007a).

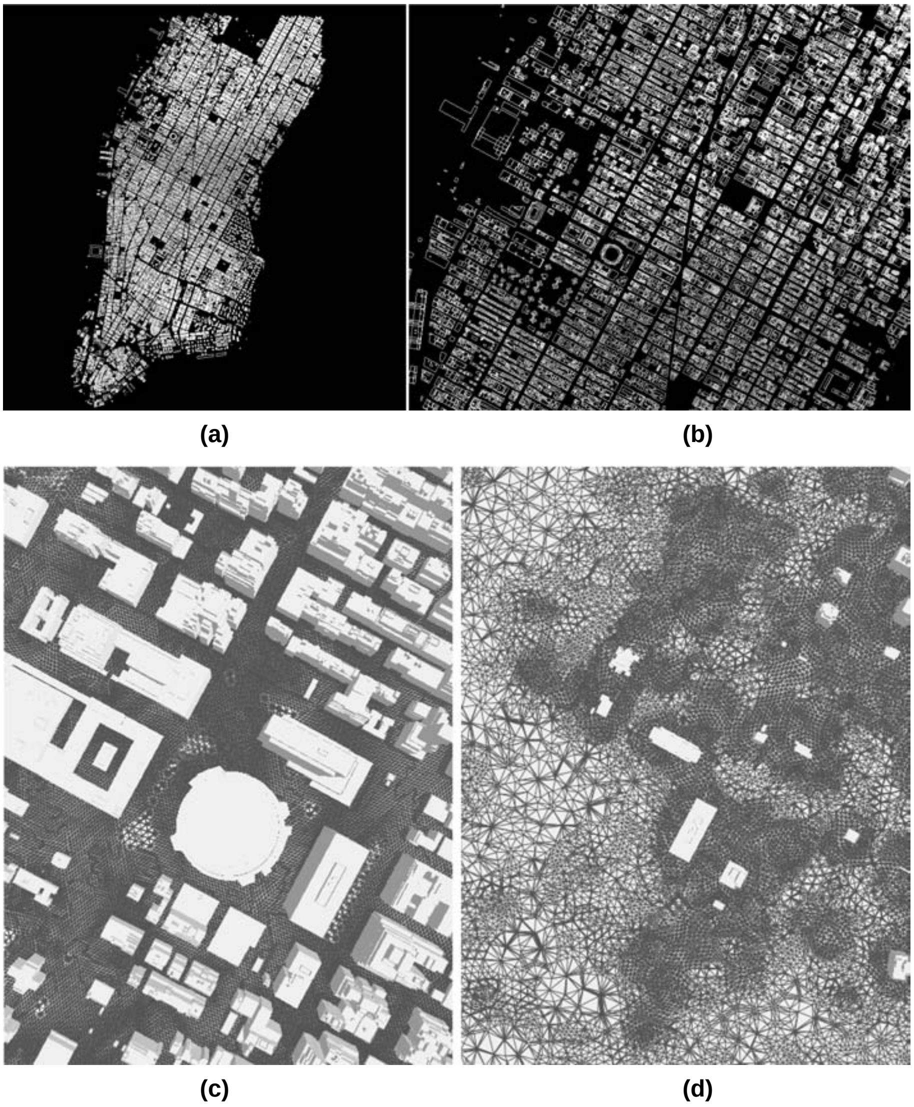
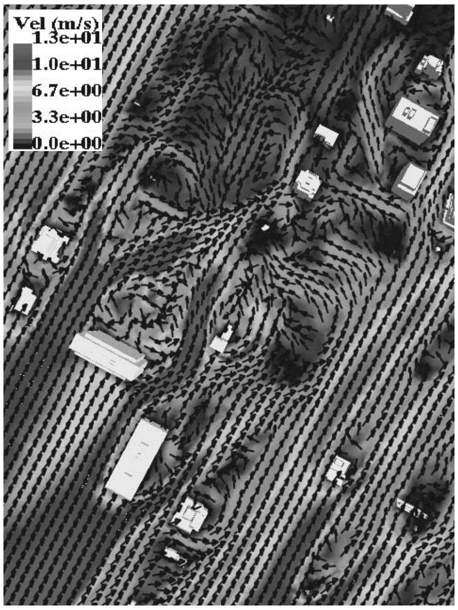
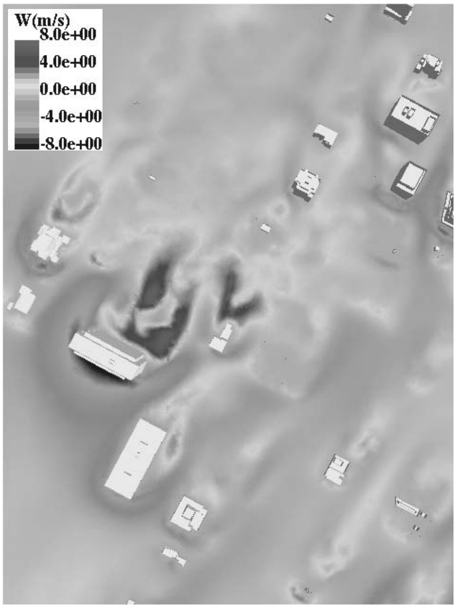


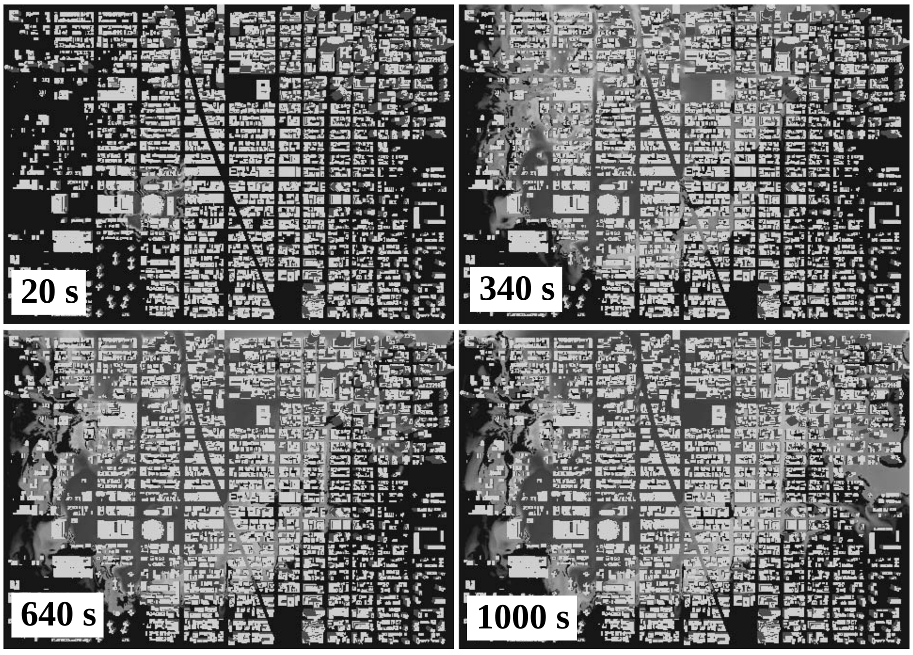
Figure 18.24. (a) Blueprint of Manhattan; (b) zoom into Madison Square Garden; cut planes at (c) 5 and (d) 100 m above ground level; SW wind case, $z = 100$ m: (e) velocity vectors and (f) contours of vertical velocity; (g)–(j): plume patterns for continuous release



(e)



(f)



(g)–(j)

Figure 18.24. Continued

18.9.6. DISPERSION IN AN INNER CITY

This case, taken from Camelli and Löhner (2006), considers the dispersion of a contaminant cloud near Madison Square Garden in New York City. The commercial building database Vexcel (see Hanna *et al.* (2006)) was used to recover the geometry description of the city. Figures 18.24(a) and (b) show the wire frame of the information contained in the database.

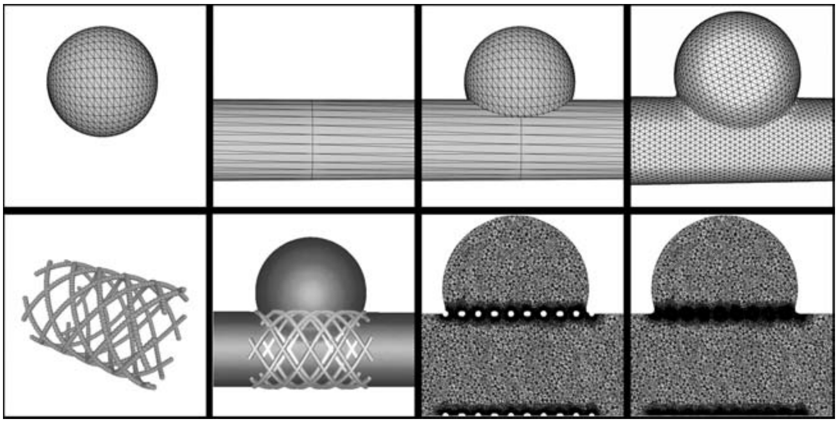
The area of buildings covered in the simulation is 2.7 km by 1.9 km. The computational domain is 3.3 km by 2.6 km and a height of 600 m. The building database simply consists of disjoined polygons that can cross and are not watertight. Cleaning up this database, even with semi-automatic tools, in order to obtain a body-fitted mesh would have taken many man-months. The adaptive embedded approach reduced dramatically the man-hours required to reconstruct the geometry of buildings, making a run like the one shown here possible. A VLES simulation was performed with a mesh of 24 million tetrahedral elements. The element size was of 2 m close to the building surfaces. Two cut planes of the volume mesh are shown in Figures 18.24(c) and (d) at 5 and 100 m above ground level. These two cut planes illustrate clearly the use of adaptive gridding based on edges crossed by the embedded surfaces, and the relatively high resolution of the mesh used close to buildings and the ground. A typical velocity field obtained for a South-Westerly wind is shown in Figures 18.24(e) and (f), and the dispersion pattern for a continuous release may be discerned from Figures 18.24(g)–(j).

18.9.7. COMPLEX ENDOVASCULAR DEVICES

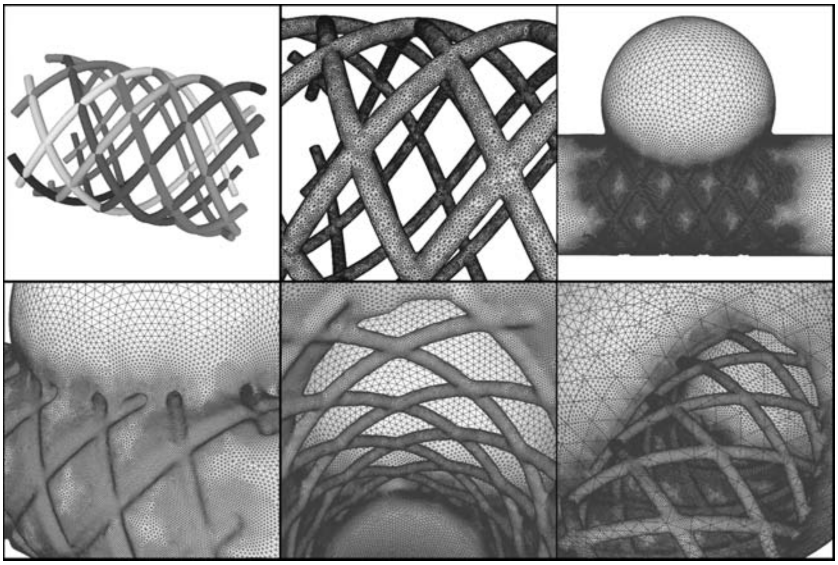
Stents are currently being considered as an interesting alternative treatment for aneurysms. While the pipeline from images to body-fitted grids to flow calculation to visualization is a rather mature process (Cebral and Löhner (2001), Cebral *et al.* (2001), Cebral and Löhner (2005)), the placement of stents or coils represents considerable challenges. In Cebral and Löhner (2005) the use of embedded grids was first proposed as a way to circumvent lengthy input times without compromising accuracy. Figure 18.25 shows the comparison of a body-fitted and embedded stent calculation for an idealized aneurysm geometry. The body-fitted idealized aneurysm is obtained by first merging a sphere with a cylinder, obtaining the isosurface of distance $\delta = 0$ (Cebral and Löhner (2001), Cebral *et al.* (2001, 2002)), and meshing this discrete surface (Figure 18.25(a)). The stent is described as a set of small spheres (beads) and is placed in the idealized aneurysm domain. The isosurface of distance $\delta = 0$ is again used as the starting point to mesh the complete domain (Figure 18.25(b)). For the embedded case, the spheres describing the stent are simply placed in their position, and the proper boundary conditions are applied as described above. Figure 18.25(c) shows the comparison of velocities for the idealized aneurysm with and without the stent. One can see that the velocities for the body-fitted and embedded approaches are nearly identical. This procedure has also been used successfully for coils (Cebral and Löhner (2005)).

18.9.8. FLOW PAST A VW GOLF 5

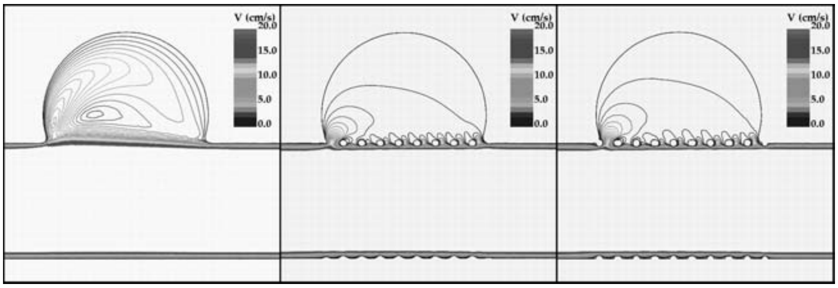
This case considers the flow past a typical passenger car, and is taken from Tilch *et al.* (2007). For external vehicle aerodynamics, the car industry is contemplating at present turnaround times of 1–2 days for arbitrary configurations. For so-called body-fitted grids, the surface definition must be watertight, and any kind of geometrical singularity, as well as small angles, should be avoided in order to generate a mesh of high quality. This typically presents no



(a)



(b)



(c)

Figure 18.25. (a) Idealized aneurysm with stent; (b) body-fitted mesh; (c) comparison of velocities in the mid-plane

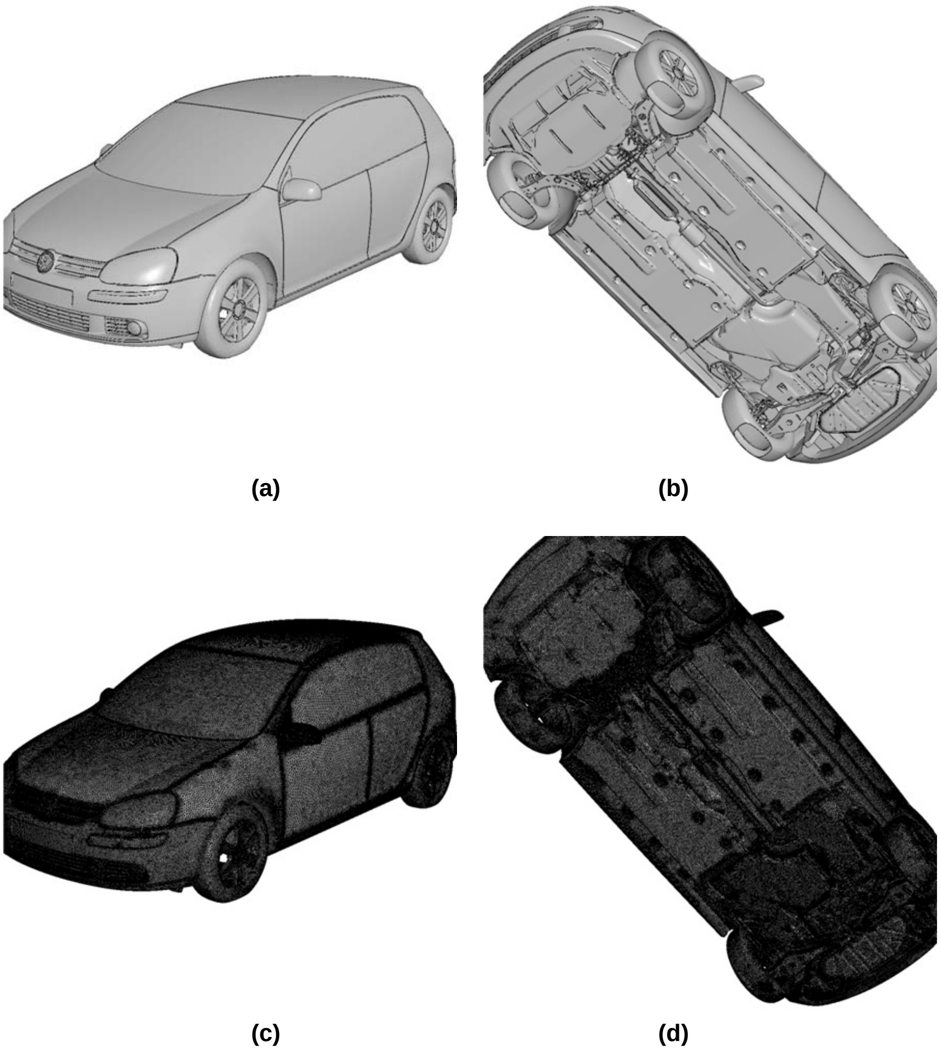
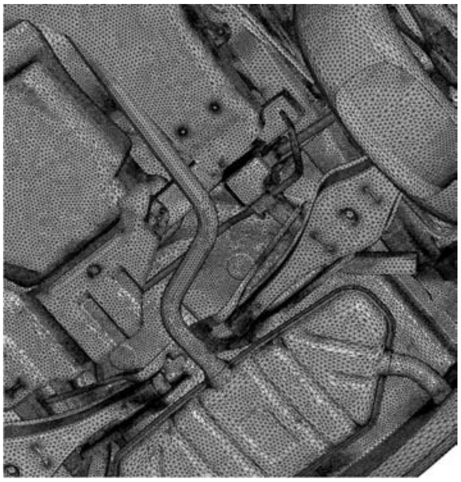
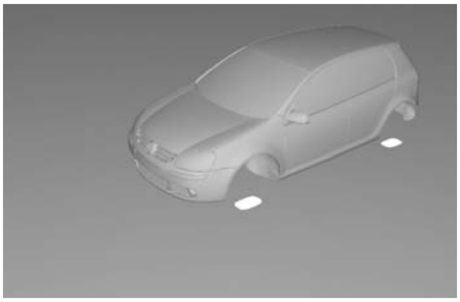


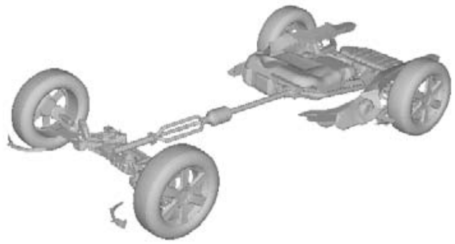
Figure 18.26. VW Golf 5: (a), (b) surface definition (body-fitted); (c), (d) surface grids (body-fitted); (e), (f) surface grids (underhood detail); (g), (h) surface grids (body-fitted, embedded); (i) surface grid (body-fitted and embedded); (j) cut mesh in the mid-plane; (k) undercarriage detail; (l) pressure (contours) on the ground and x-planes, velocity (shaded) in the back x-plane; (m) velocities in mid-plane (top, body-fitted; bottom, body-fitted + embedded)

problems for the main 'shape' of the car (the part visible to a streetside observer), but can be difficult to obtain in such a short time for the underhood and undercarriage of a typical car or truck. Experience indicates that, even with sophisticated software toolkits, manual cleanup in most cases takes several days for a complete car. At first sight, the solution of high-Reynolds-number flows with grids of this type seems improper. Indeed, for the external shape portion the surface is smooth, and the interplay of pressure gradient and viscous/advective terms

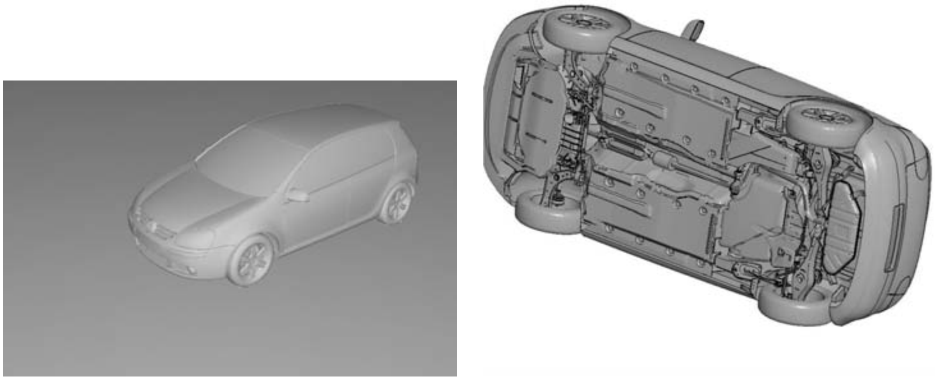


(e), (f)



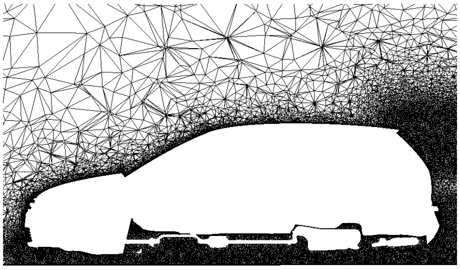


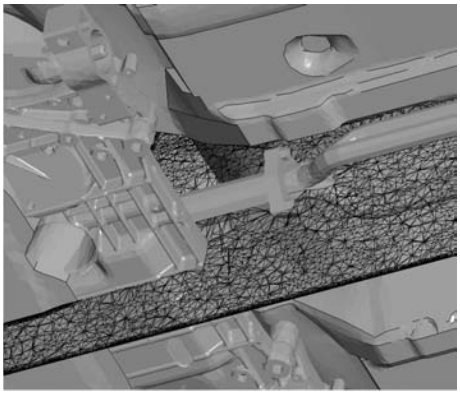
(g) (h)



(i)

Figure 18.26. Continued


(j)


(k)

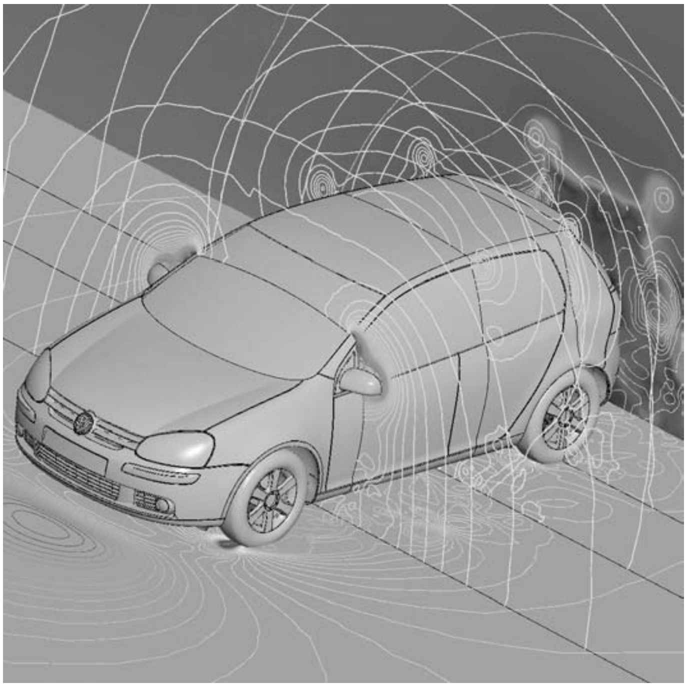
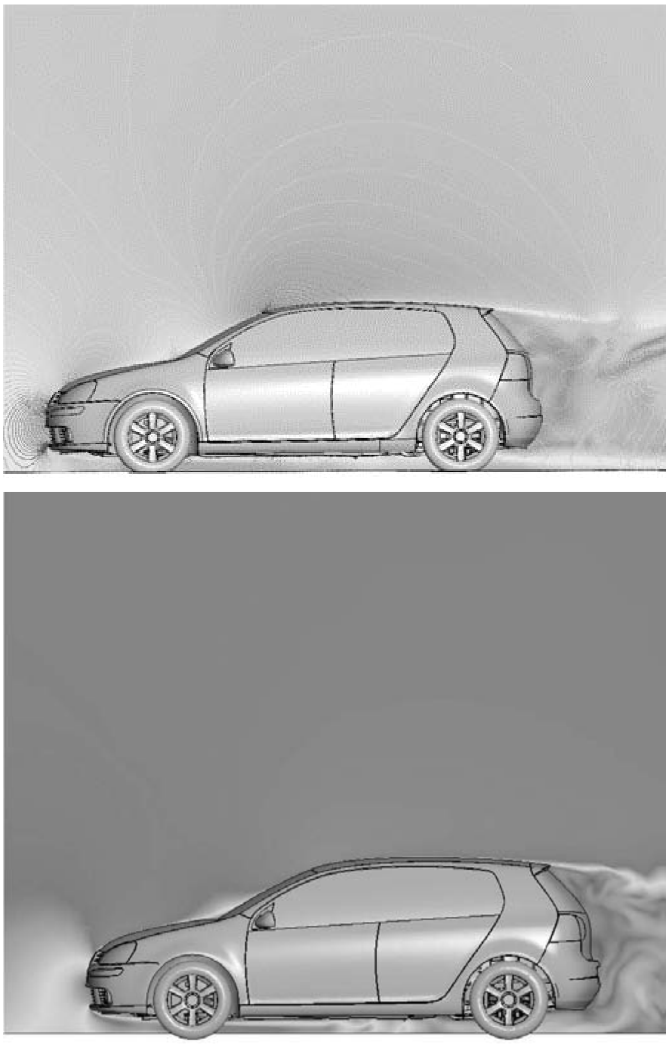

(l)

Figure 18.26. Continued

is what decides whether separation will occur. Therefore, for this portion of the vehicle, a highly detailed, body-fitted RANS is considered mandatory. However, for the underhood and undercarriage, many abrupt changes in curvature occur, the flow is massively separated and an LES run seems sufficient. For embedded grids, this presents no problem. One is therefore in a rather fortunate position: the region where the geometry is the ‘dirtiest’ happens to be the region where isotropic grids are sufficient, making this region a good candidate for embedded



(m)

Figure 18.26. Continued

grids. The key idea is then to obtain, quickly, the external shape of the vehicle and grid it with typical body-fitted RANS grids. Note that this portion of the surface is typically ‘clean’, i.e. a turnaround of 1–2 days is possible. The underhood and undercarriage, on the other hand, are then inserted into the RANS mesh generated for the external shape of the vehicle as an embedded surface. As such, it can have crossing faces (stemming, for example, from different parts of the undercarriage), and does not require elements of very high quality. A run is then conducted with the embedded mesh.

The example shown here compares a body-fitted and an embedded run of the type described above for a typical passenger car. The body-fitted mesh was obtained after several weeks of cleanup, and may be seen in Figures 18.26(a) and (b). Note that all the undercarriage details have been taken into account. The corresponding surface grids are shown in Figures 18.26(c)–(e). For the embedded case, the starting point was given by two NASTRAN files which came from different departments at VW. The surface which was meshed using the body-fitted approach was given by 12 patches (one surface for the top, one for the bottom, two for the mirrors), whereas the surface which was treated with the embedded approach was given by 106 parts, most of which were single patches. The complete surface triangulation used to define the car had 1.1 million triangles. The body-fitted mesh consisted of approximately 5.68 million points and 32.03 million tetrahedra. Five RANS layers were used. For the car body, the first point was $y_w = 0.758$ mm away from the wall, and the mesh was increased by a factor of $c_i = 1.5$ between layers. For the ground, the first point was $y_w = 0.914$ mm away from the ground, and the mesh was increased by a factor of $c_i = 1.4$ between layers. The surface of the body-fitted domain, the embedded surface, as well as the complete vehicle, are shown in Figures 18.26(f)–(i). A close-up of the undercarriage, together with the mesh in a cut plane, is shown in Figures 18.26(j). The physical parameters were set as $\mathbf{v}_\infty = (33.33, 0, 0)$ m/s, $\rho = 1.2$ kg/m^3, $\mu = 1.4 \times 10^{-5}$ kg/m, yielding a Reynolds number of approximately $Re = 10^7$. A Smagorinsky turbulence model was used. The run was initialized with approximately 10^3 timesteps using local timesteps. This was followed by a time-accurate run of 10^4 timesteps, integrating explicitly the advective terms in order to obtain an accurate wake. The results obtained are shown in Figures 18.26(k)–(m).

The drag coefficient obtained was $c_d = 0.309$, based on a reference velocity of $v_\infty = 33.33$ m/s and an area of $A = 2.2$ m^2. Experiments conducted at VW measured a drag coefficient of $c_d = 0.330$. However, the wind-tunnel model exhibited an open grille. From past experience, one can infer that performing the experiment with a closed front, as was done for the present run, would reduce the c_d by 5–10% in comparison with an open grille. At VW, the estimated value for the closed grille case was $c_d = 0.305$. The purely body-fitted CFD run with the same code yielded a drag coefficient of $c_d = 0.320$. Overall, this leads to the conclusion that the combined body-fitted/embedded approach leads to results that are within 5% of experimental values, impressive considering the reduction in set-up times, and well within a range to render them interesting to designers. Moreover, as experience with this approach accumulates, one may reasonably expect to be able to obtain even better results.

19 TREATMENT OF FREE SURFACES

Many problems in science and engineering are characterized by the movement of free surfaces. The most familiar are those where the movement of the liquid–gas interface leads to waves. In fact, for most people the notion of fluid dynamics is immediately associated with the motion of waves. Waves in high sea states, waves breaking near shores and moving ships, the interaction of extreme waves with floating structures, the spillage of water on ships or offshore structures (so-called green water on deck) and sloshing are but a few examples of flows with free surface motion. Other applications include:

- in industrial engineering: melts, stirring tanks, mould filling, extrusion, etc.;
- in aerospace engineering: sloshing in the fuel tanks of satellites, water landings, etc.

The computation of free surface flows is difficult because neither the shape nor the position of the interface between gas and liquid is known *a priori*; on the contrary, it often involves unsteady fragmentation and merging processes. There are basically three approaches to the computation of flows with free surfaces:

- interface fitting;
- interface tracking; and
- interface capturing methods.

In the *interface fitting* methods the free surface is treated as a boundary of the computational domain, where the kinematic and dynamic boundary conditions are applied. These methods cannot be used if the interface topology changes significantly, particularly if the problem leads to overturning or breaking waves. In the *interface tracking* methods the free surface is represented and tracked explicitly either by marking it with special marker points, or by moving it as an embedded surface inside a volume mesh. *Interface capturing* methods consider both fluids as a single effective fluid with variable properties; the interface is captured as a region of sudden change in fluid properties. While more general, it is less accurate than the interface fitting methods.

19.1. Interface fitting methods

This class of methods is best explained for the case of wave resistance of ships, where it has been developed to a high degree of maturity. In order to fix the notation, let us consider the ship shown in Figure 19.1. The Cartesian coordinate system $Oxyz$ is fixed to the ship with

Applied Computational Fluid Dynamics Techniques: An Introduction Based on Finite Element Methods, Second Edition.
Rainald Löhner © 2008 John Wiley & Sons, Ltd. ISBN: 978-0-470-51907-3

the origin inside the hull on the mean free surface. The z-direction is positive upwards, y is positive towards the starboard side and x is positive in the aft direction. The free stream velocity vector is parallel to the x-axis and points in the same direction. While the use of such a coordinate system may be considered restrictive, it is general, and other problems, requiring other coordinate systems, can be treated in a similar way.

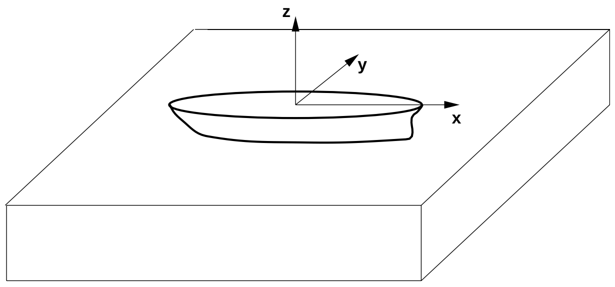


Figure 19.1. Coordinate system used

The water is described by the incompressible RANS equations, given, in non-dimensional form, by the conservation of mass and momentum:

$$\nabla \cdot \mathbf{v} = 0, \tag{19.1}$$

$$\rho \mathbf{v}_{,t} + \rho \mathbf{v} \cdot \nabla \mathbf{v} + \nabla \Psi = \nabla \mu \nabla \mathbf{v}, \tag{19.2}$$

where $\mathbf{v} = (u, v, w)$ denotes the velocity vector and Ψ the pressure plus hydrostatic pressure:

$$\Psi = p + \rho g z. \tag{19.3}$$

A particle on the free surface must remain so for all times, which implies that the free surface elevation β obeys the advection equation

$$\beta_{,t} + u\beta_{,x} + v\beta_{,y} = w. \tag{19.4}$$

The boundary conditions are as follows.

(a) *Inflow plane*. At the inflow plane, the velocity, pressure and free surface height are prescribed:

$$\mathbf{v} = (u_\infty, 0, 0), \ \Psi = 0, \ \beta = 0. \tag{19.5}$$

(b) *Exit plane*. At the exit plane, none of the quantities are prescribed. The natural Neumann conditions for the pressure and extrapolation boundary conditions for the velocities and free surface height are imposed automatically by the numerical scheme used.

(c) *Free surface*. At the free surface, the pressure p is prescribed to be $p = 0$, implying that Ψ is given by

$$\Psi = \rho g \beta. \tag{19.6}$$

The velocity is allowed to float freely, allowing leakage of fluid through the free surface.

(d) *Bottom*. At the bottom, one may either impose the boundary conditions for:

- *a wall*: vanishing normal velocity, Neumann conditions for the pressure, or
- *an infinite depth*: prescribed pressure, no boundary conditions for the velocities.

(e) *Ship hull*. Depending on the turbulence model used, the velocity (laminar, Baldwin–Lomax, low-$Re\ k - \epsilon$) or only its normal component (Euler, high-$Re\ k - \epsilon$) must vanish, i.e.

$$\mathbf{v} = 0, \quad \mathbf{v} \cdot \mathbf{n} = 0, \tag{19.7a,b}$$

where $\mathbf{n}$ is the normal to the hull.

(f) *Side walls*. On the side walls of the computational domain, a vanishing normal velocity is imposed. For the integration of (19.1), (19.2) and (19.4) any of the schemes described in Chapter 11 may be employed.

19.1.1. FREE SURFACE DISCRETIZATION

The free surface equation (19.4) is treated as a standard scalar advection equation with source terms for the x, y plane. The faces on the free surface are extracted from the 3-D volume grid. Points and elements are renumbered locally to obtain a 2-D triangular finite element mesh in x, y. For an edge-based solver, the spatial discretization of the advective fluxes results in

$$\mathcal{F}_{ij} = \mathbf{f}_i + \mathbf{f}_j, \quad \mathbf{f}_i = (S_x^{ij} u_i + S_y^{ij} v_i) \cdot \beta_i. \tag{19.8}$$

Fourth-order damping is added to stabilize these central difference terms, resulting in

$$\mathcal{F}_{ij} = \mathbf{f}_i + \mathbf{f}_j - |\lambda^{ij}| \left(\beta_i - \beta_j + \frac{l^{ij}}{2} (\nabla \beta_i + \nabla \beta_j) \right). \tag{19.9}$$

In order to avoid spurious reflections at the boundaries, damping terms are added to the basic transport equation (19.4). The most common of these was proposed by Hino *et al.* (1993), and is given by

$$\beta_{,t} + u\beta_{,x} + v\beta_{,y} = w - d_h(\mathbf{x})\beta, \tag{19.10}$$

where $d_h(\mathbf{x})$, for the outflow boundary, is given by

$$d_h = c_1 \xi^2, \quad \xi = \max\left(0, \frac{x - x_{d\,\max}}{x_{\max} - x_{d\,\max}}\right),$$
$$x_{d\,\max} = x_{\max} - 2\pi Fr^2 L, \tag{19.11}$$

where $Fr = |\mathbf{v}_\infty| / \sqrt{gL}$ denotes the Froude number, L is the characteristic length (e.g. the length of the ship), $|\mathbf{v}_\infty|$ the velocity of the ship and c_1 is a parameter of $O(1)$. A similar expression is applied at the inflow boundary for steady-state problems. The vertical velocity w, as well as the additional damping term, are evaluated by simply using the lumped mass matrix. In order to damp out the wave height β completely at the downstream boundary, the w-velocity obtained from the 3-D incompressible flow solver is modified, yielding

$$\beta_{,t} + u\beta_{,x} + v\beta_{,y} = d_w(\mathbf{x})w - d_h(\mathbf{x})\beta, \tag{19.12}$$

where $d_w(\mathbf{x})$ is given by the Hermitian polynomial (Löhner (2001))

$$d_w = 1 - 3\xi^2 + 2\xi^3, \tag{19.13}$$

and ξ is defined in (19.11). The final semi-discrete scheme takes the form

$$\mathbf{M}_l \beta_{,t} = \mathbf{r} = \mathbf{r}_a(u, v, \beta) + \mathbf{r}_s(d_w, w) + \mathbf{r}_d(d_h, \beta), \tag{19.14}$$

where the subscripts a, s and d stand for advection, source and damping. This system of ODE's is integrated in time using explicit time-marching schemes, e.g. a standard five-stage Runge–Kutta scheme.

19.1.2. OVERALL SCHEME

One complete timestep consists of the following steps:

- given the boundary conditions for the pressure Ψ, update the solution in the 3-D fluid mesh (velocities, pressures, turbulence variables, etc.);
- extract the velocity vector $\mathbf{v} = (u, v, w)$ at the free surface and transfer it to the 2-D free surface module;
- given the velocity field, update the free surface β;
- transfer back the new free surface β to the 3-D fluid mesh, and impose new boundary conditions for the pressure Ψ.

For steady-state applications, the fluid and free surface domains are updated using local timesteps. This allows some room for variants that may converge faster to the final solution, e.g. n steps of the fluid followed by m steps of the free surface, complete convergence of the free surface between fluid updates, etc. Empirical evidence (Löhner *et al.* (1998, 1999a,c)) indicates that most of these variants prove unstable, or do not accelerate convergence measurably. For steady-state applications it was found that an equivalent 'time-interval' ratio between the fluid and the free surface of 1:8 yielded the fastest convergence (e.g. a Courant number of $C_f = 0.25$ for the fluid and $C_s = 2.0$ for the free surface).

19.1.3. MESH UPDATE

Schemes that work with structured grids (e.g. Hino (1989, 1997), Hino *et al.* (1993), Farmer *et al.* (1993), Martinelli and Farmer (1994), Cowles and Martinelli (1996)) march the solution in time until a steady state is reached. At each timestep, a volume update is followed by a free surface update. The repositioning of points at each timestep implies a complete recalculation of geometrical parameters, as well as interrogation of the CAD information defining the surface. For general unstructured grids, this can lead to a doubling of CPU requirements. For this reason, when solving steady-state problems, it is advisable not to move the grid at each timestep, but only change the pressure boundary condition after each update of the free surface β. The mesh is updated every 100 to 250 timesteps, thereby minimizing the costs associated with geometry recalculations and grid repositioning along surfaces. One can also observe that this strategy has the advantage of not moving the mesh unduly at the beginning of a run, where large wave amplitudes may be present. One mesh update consists of the following steps.

- Obtain the new elevation for the points on the free surface from β. This only results in a vertical (z-direction) displacement field $\mathbf{d}_\Gamma$ for the boundary points.
- Apply the proper boundary conditions for the points on the waterline. This results in an additional horizontal (x,y-direction) displacement field for the points on the water line.
- Smooth the displacement field in order to avoid mesh distortion. This may be accomplished with any of the techniques described in Chapter 12.
- Interrogate the CAD data to reposition the points on the hull.

Denoting by $\mathbf{d}_*$, $\mathbf{n}$ and $\mathbf{t}$ the predicted displacement of each point, surface normals and tangential directions, the boundary conditions for the mesh movement are as follows (see Figure 19.2).

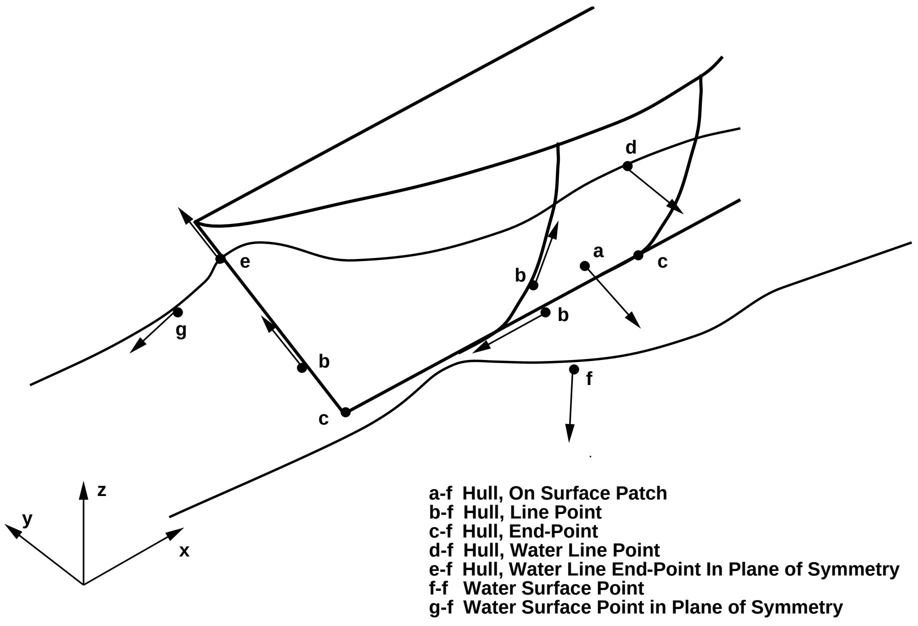

d
a
c
e
b
b
g
b
c
f
z
y
x
a-f Hull, On Surface Patch
b-f Hull, Line Point
c-f Hull, End-Point
d-f Hull, Water Line Point
e-f Hull, Water Line End-Point In Plane of Symmetry
f-f Water Surface Point
g-f Water Surface Point in Plane of Symmetry

Figure 19.2. Boundary conditions for mesh movement

(a) *Hull, on surface patch*. The movement of these points has to be along the surface, i.e. the normal component of $\mathbf{d}_*$ is removed:

$$\mathbf{d} = \mathbf{d}_* - (\mathbf{d}_* \cdot \mathbf{n})\,\mathbf{n}. \tag{19.15}$$

(b) *Hull, line point*. The movement of these points has to be along the lines, resulting in a tangential boundary displacement of the form

$$\mathbf{d} = (\mathbf{d}_* \cdot \mathbf{t})\mathbf{t}. \tag{19.16}$$

(c) *Hull, endpoint.* No displacement is allowed for these points, i.e. $\mathbf{d} = 0$.

(d) *Hull/water line point, water line endpoint.* The displacement of these points is fixed, given by the change in elevation Δz and the surface normal of the hull. Defining $\mathbf{d}_0 = (0, 0, \Delta z)$, we have

$$\mathbf{d} = \frac{\mathbf{d}_0 - (\mathbf{d}_0 \cdot \mathbf{n})\,\mathbf{n}}{1 - n_z^2}. \tag{19.17}$$

(e) *Hull/water line endpoint in plane of symmetry.* The displacement of these points is fixed, and dictated by the tangential vector to the hull line in the symmetry plane:

$$\mathbf{d} = \frac{(\mathbf{d}_0 \cdot \mathbf{t})\mathbf{t}}{1 - n_t^2}. \tag{19.18}$$

(f) *Water surface points.* These points start with an initial displacement $\mathbf{d}_0$, but may glide along the water surface, allowing the mesh to accommodate the displacements in the x,y-directions due to points on the hull. The normal to the water line is taken, and (19.17) is used to correct any further displacements.

(g) *Water surface points in plane of symmetry.* As before, these points start with an initial displacement $\mathbf{d}_0$, but may glide along the water surface, remaining in the plane of symmetry, thus allowing the mesh to accommodate the displacements in the x-direction due to points on the hull. The tangential direction is obtained from the sides lying on the water surface in the plane of symmetry, and (19.18) is used to correct any further displacements.

An option to restrict the movement of points completely in ‘difficult’ regions of the mesh is often employed. Regions where such an option is required are transom sterns, as well as the points lying in the half-plane given by the minimum z-value of the hull. Should negative elements arise due to surface point repositioning, they are removed and a local remeshing takes place. Naturally, these situations should be avoided as much as possible.

19.1.4. EXAMPLES FOR SURFACE FITTING

We include some examples that were computed using the algorithm outlined in the preceeding sections. All of these consider the prediction of steady wave patterns for hulls over a wide range of Froude numbers. For all of these cases, local timestepping was employed for the 3-D incompressible flow solvers as well as the free surface solver. At the start of a run, the 3-D flowfield was updated for 10 timesteps without any free surface update. Thereafter, the free surface was updated after every 3-D flowfield timestep. The mesh was moved every 100 to 250 timesteps.

19.1.4.1. Submerged NACA0012

The first case considered is a submerged NACA0012 at $\alpha = 5°$ angle of attack. This same configuration was tested experimentally by Duncan (1983) and modelled numerically by Hino *et al.* (1993), Hino (1997). Although the case is 2-D, it was modelled as 3-D, with two parallel walls in the y-direction. The mesh consisted of 2 409 720 tetrahedral elements,

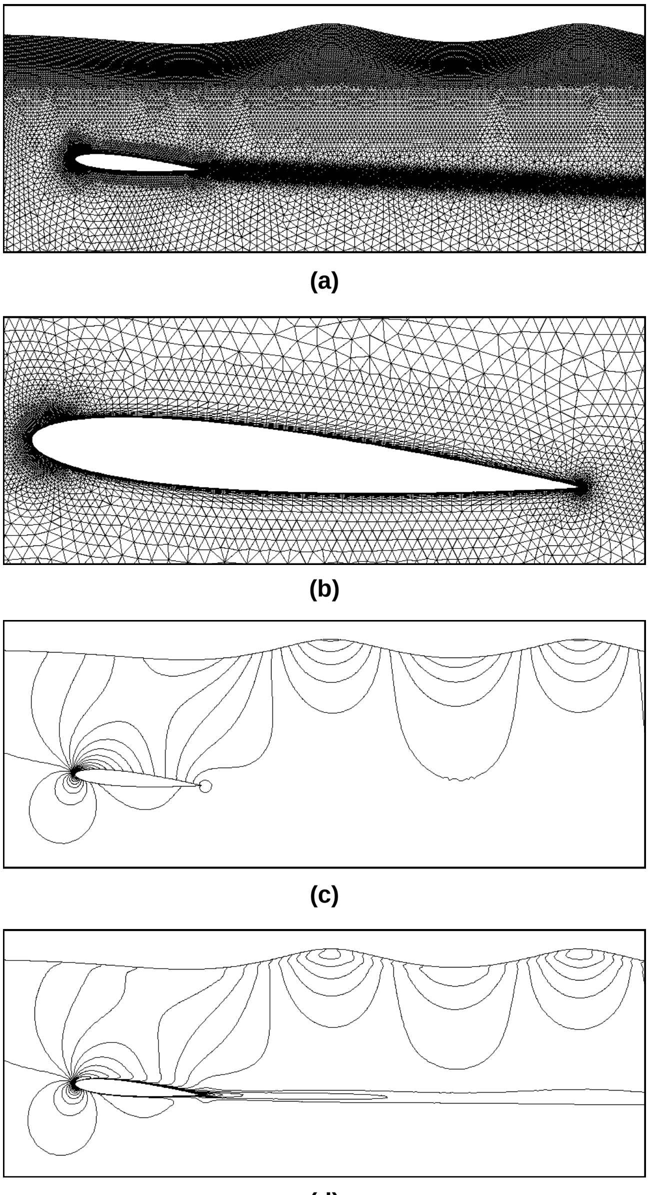
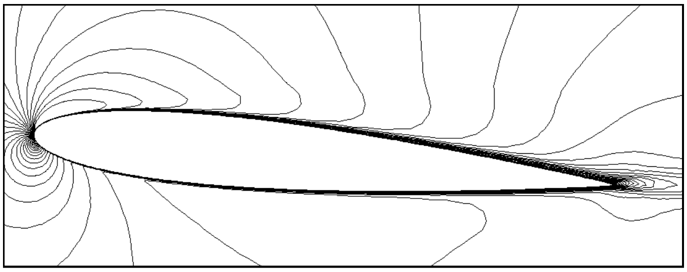
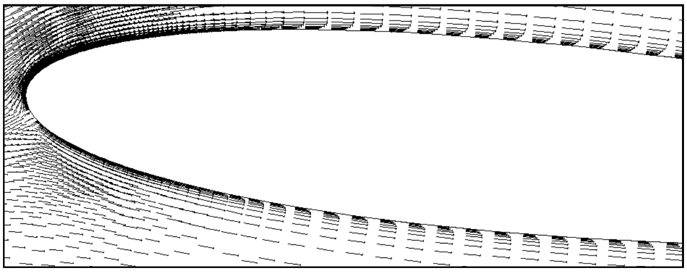


Figure 19.3. Submerged NACA0012: (a), (b) surface grids; (c), (d) pressure and velocity fields; (e), (f) velocity field (zoom); (g) wave profiles

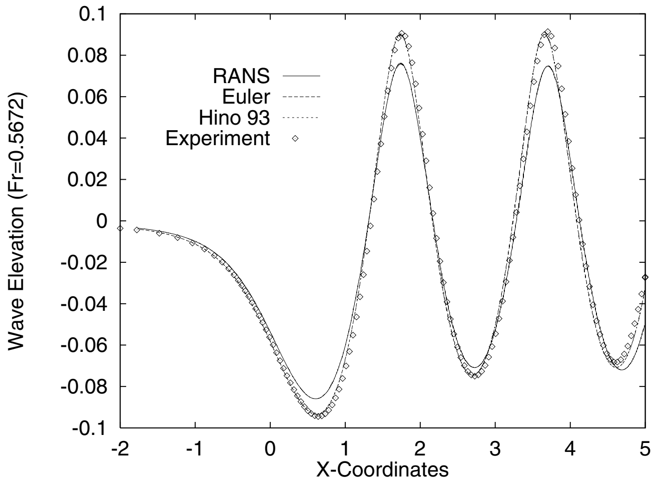
465 752 points and 11 093 boundary points, of which 6929 were on the free surface. The Froude number was set to $Fr = 0.5672$. This case was run in Euler mode and using the Baldwin–Lomax model with a Reynolds number of $Re = 10^6$. Figures 19.3(a)–(f) show the surface grid, pressure and velocity fields, as well as a zoom of the velocity field close to



(e)



(f)



(g)

Figure 19.3. Continued

the airfoil. Note the boundary layer from the velocity fields. Figure 19.3(g) compares the wave profiles for the Euler, Baldwin–Lomax, Hino *et al.*'s (1993) Euler and Duncan's (1983) experiment data. The wave amplitudes are noticeably lower for the RANS case. Interestingly, this was also observed by Hino (1997).

19.1.4.2. Wigley hull

The next case is the well-known Wigley hull, given by the analytical formula

$$y = 0.5 \cdot B \cdot [1 - 4x^2] \cdot \left[1 - \left(\frac{z}{D}\right)^2\right], \tag{19.19}$$

where B and D are the beam and the draft of the ship at still water. For the case considered here, $D = 0.0625$ and $B = 0.1$. This same configuration was tested experimentally at the University of Tokyo (ITTC (1983a, b)) and modelled numerically by Farmer *et al.* (1993), Raven (1996) and others. At first, a fine triangulation for the surface given by (19.19) was generated. This triangulation was subsequently used to define, in a discrete manner, the hull. The surface definition of the complete computational domain consisted of discrete (hull) and analytical surface patches. The mesh consisted of 1 119 703 tetrahedral elements, 204 155 points and 30 358 boundary points, of which 15 515 were on the free surface. The parameters for this simulations were as follows: $Fr = 0.25$, $Re = 10^6$ and the $k - \epsilon$ model with the law of the wall approximation. Figures 19.4(a) and (b) show the surface grids employed, and the extent of the region with high-aspect-ratio elements. In Figures 19.4(c) and (d) the wave profiles and surface velocities obtained from Euler and RANS calculations are compared. As expected, the effect of viscosity becomes noticeable in the stern region. Figure 19.4(e) shows the comparison to the experiments conducted at the University of Tokyo. It can be noticed that the first wave is accurately reproduced, but that the second wave is not well reproduced by the calculations.

19.1.4.3. Double Wigley hull

The third case considered is an inviscid case comprising two Wigley hulls positioned close to each other. Figures 19.5(a)–(f) show the resulting wave pattern for a Froude number of $Fr = 0.316$ for different relative spacings in the x- and y-directions. As one can see, the effect on the resulting wave pattern is considerable.

19.1.4.4. Wigley carrier group

The next case considered is again inviscid. The configuration is composed of a Wigley hull in the centre that has been enlarged by a factor of 1:3, surrounded by normal Wigley hulls. The mesh consisted of approximately 4.2 million tetrahedra. Figures 19.6(a) and (b) show the resulting wave pattern for a Froude number of $Fr = 0.316$. The CPU time for this problem was approximately 24 hours using eight processors on an SGI Origin2000.

19.1.5. PRACTICAL LIMITATIONS OF FREE SURFACE FITTING

While very accurate and extremely competitive in terms of storage and CPU requirements as compared to other methods, free surface fitting also has limitations. The key limitation stems from the free surface description given by (19.4). Any free surface described in this manner can only be single-valued in the z-direction. Therefore, it will be impossible to describe a breaking wave. Another case where this method fails is the situation where transom sterns have vertical walls. Here, as before, the free surface becomes multi-valued. Summarizing, free surface fitting methods cannot be used if the interface topology changes significantly.

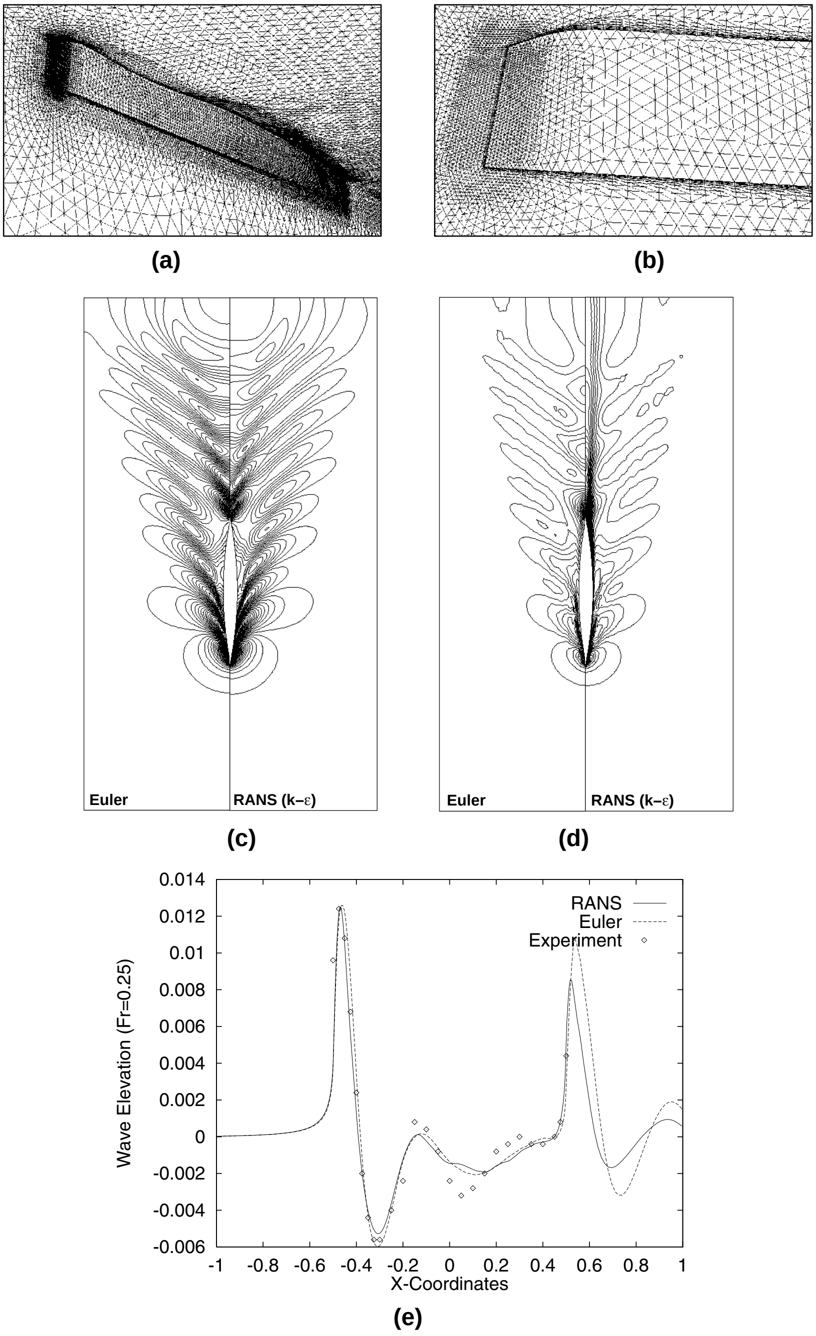


Figure 19.4. Wigley hull: (a), (b) surface of mesh; (c) wave elevation; and (d) surface velocity; (e) wave elevation at the hull

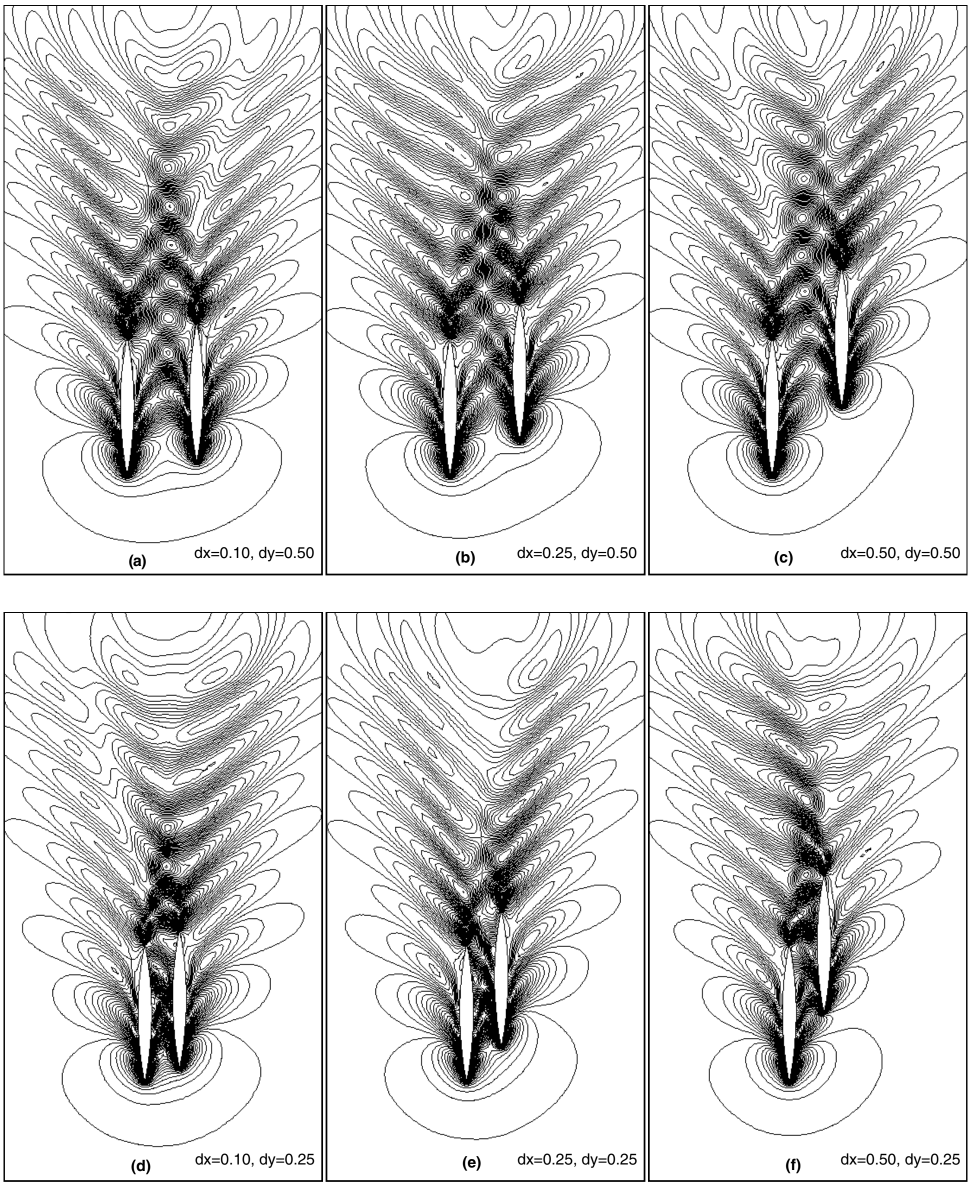
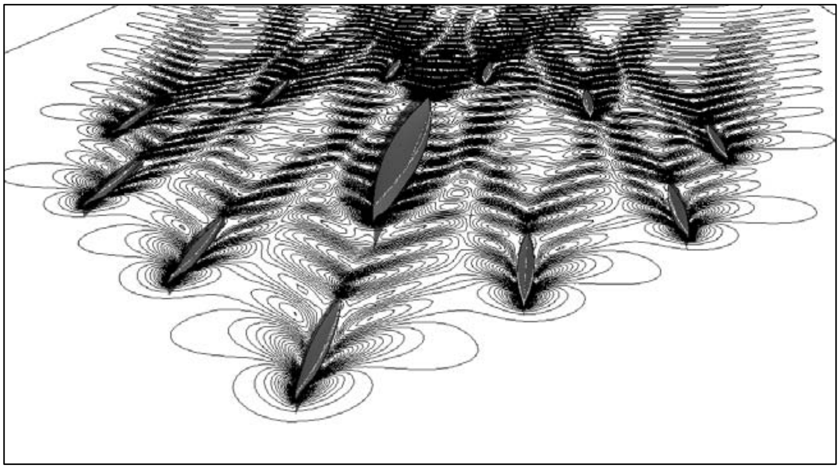


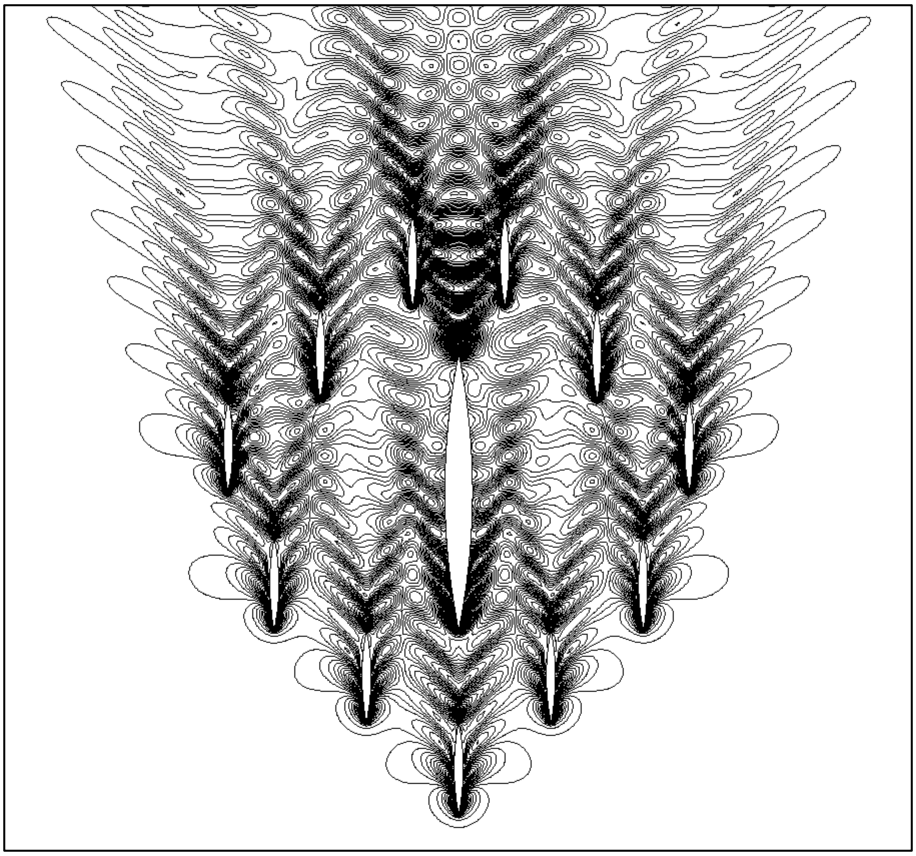
Figure 19.5. (a)–(c) Wave elevation for two Wigley hulls ($Fr = 0.316$); (d)–(f) wave elevation for two Wigley hulls ($Fr = 0.316$)

19.2. Interface capturing methods

As stated before, the third possible approach to treat free surfaces is given by the so-called interface-capturing methods (Nichols and Hirt (1975), Hirt and Nichols (1981), Yabe and Aoki (1991), Unverdi and Tryggvason (1992), Sussman *et al.* (1994), Yabe (1997),



(a)



(b)

Figure 19.6. (a) Wigley carrier group; (b) wave elevation ($Fr = 0.316$)

Scardovelli and Zaleski (1999), Chen and Kharif (1999), Fekken *et al.* (1999), Enright *et al.* (2003), Biausser *et al.* (2004), Huijsmans and van Grosen (2004), Coppola-Owen and Codina (2005)). These consider both fluids as a single effective fluid with variable properties; the interface is captured as a region of sudden change in fluid properties. The main problem of complex free surface flows is that the density ρ jumps by three orders of magnitude between the gaseous and liquid phases. Moreover, this surface can move, bend and reconnect in arbitrary ways. In order to illustrate the difficulties that can arise if one treats the complete system, consider a hydrostatic flow, where the exact solution is $\mathbf{v} = 0$, $p = -\rho\mathbf{g} \cdot (\mathbf{x} - \mathbf{x}_0)$, and $\mathbf{x}_0$ denotes the position of the free surface. Unless the free surface coincides with the faces of elements, there is no way for typical finite element shape functions to capture the discontinuity in the gradient of the pressure. This implies that one has to either increase the number of Gauss points (Codina and Soto (2002)) or modify (e.g. enrich) the shape function space (Coppola-Owen and Codina (2005), Kölke (2005)). Using the standard linear element procedure leads to spurious velocity jumps at the interface, as any small pressure gradient that ‘pollutes over’ from the water to the air region will accelerate the air considerably. This in turn will lead to loss of divergence, causing more spurious pressures. The whole cycle may, in fact, lead to a complete divergence of the solution. Faced with this dilemma, most flows with free surfaces have been solved neglecting the air. This approach does not account for the pressure buildup due to volumes of gas enclosed by liquid, and therefore is not universal.

The liquid–gas interface is described by a scalar equation of the form

$$\Phi_{,t} + \mathbf{v}_a \cdot \nabla\Phi = 0. \tag{19.20}$$

For the classic volume of fluid (VOF) technique, Φ represents the percentage of liquid in a cell/element or control volume (see Nichols and Hirt (1975), Hirt and Nichols (1981), Scardovelli and Zaleski (1999), Chen and Kharif (1999), Fekken *et al.* (1999), Biausser *et al.* (2004), Huijsmans and van Grosen (2004)). For pseudo-concentration (PC) techniques, Φ represents the total density of the material in a cell/element or control volume. For the level set (LS) approach Φ represents the signed distance to the interface (Enright *et al.* (2003)).

One *complete timestep* for a projection-based incompressible flow solver as described in Chapter 11 then comprises of the following substeps:

- predict velocity (advective-diffusive predictor, equations (11.32a), (11.41) and (11.42));
- extrapolate the pressure (imposition of boundary conditions);
- update the pressure (equation (11.32b));
- correct the velocity field (equation (11.32c));
- extrapolate the velocity field; and
- update the scalar interface indicator.

The extension of a solver for the incompressible Navier–Stokes equations to handle free surface flows via the VOF or LS techniques requires a series of extensions which are the subject of the present section. At this point, we remark that the implementation of the VOF and LS approaches is very similar. Moreover, experience indicates that both work well.

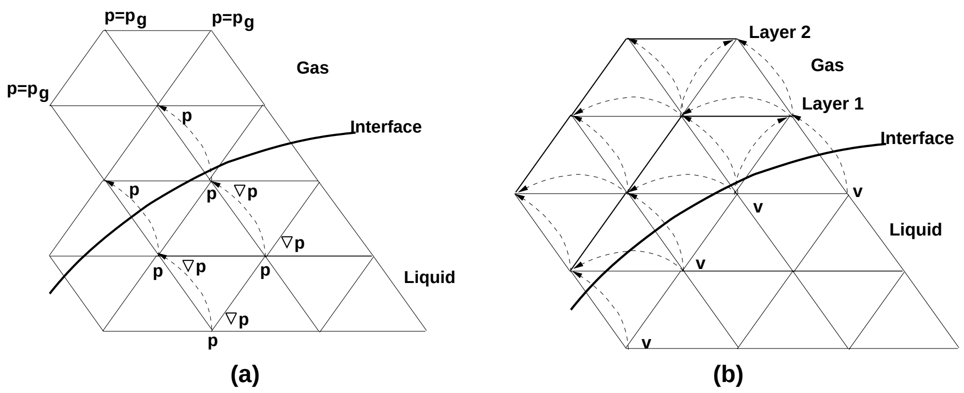


Figure 19.7. Extrapolation of (a) the pressure and (b) the velocity

For VOF, it is important to have a monotonicity preserving scheme for Ψ. For LS, it is important to balance the cost and accuracy loss of re-initializations *vis-à-vis* propagation. In what follows, we will assume that Φ is bounded by values for liquid and gas (e.g. $0 \leq \Phi \leq 1$ for VOF, $\rho_g \leq \Phi \leq \rho_l$ for PC) and that the liquid–gas interface is defined by the average of these extreme values (i.e. $\Phi = 0.5$ for VOF, $\Phi = 0.5 \cdot (\rho_g + \rho_l)$ for PC, $\Phi = 0$ for LS).

19.2.1. EXTRAPOLATION OF THE PRESSURE

The pressure in the gas region needs to be extrapolated in order to obtain the proper velocities in the region of the free surface. This extrapolation is performed using a three-step procedure. In the first step, the pressures for all points in the gas region are set to (constant) values, either the atmospheric pressure or, in the case of bubbles, the pressure of the particular bubble. In a second step, the gradient of the pressure for the points in the liquid that are close to the liquid–gas interface are extrapolated from the points inside the liquid region (see Figure 19.7(a)). This step is required as the pressure gradient for these points cannot be computed properly from the data given. Using this information (i.e. pressure and gradient of pressure), the pressure for the points in the gas that are close to the liquid–gas interface are computed.

19.2.2. EXTRAPOLATION OF THE VELOCITY

The velocity in the gas region needs to be extrapolated properly in order to propagate accurately the free surface. This extrapolation is started by initializing all velocities in the gas region to $\mathbf{v} = 0$. Then, for each subsequent layer of points in the gas region where velocities have not been extrapolated (unknown values), an average of the velocities of the surrounding points with known values is taken (see Figure 19.7(b)).

19.2.3. KEEPING INTERFACES SHARP

The VOF and PC options propagate Heavyside functions through an Eulerian mesh. The 'sharpness' of such profiles requires the use of monotonicity-preserving schemes for advection, such as total variation diminishing (TVD) or flux-corrected transport (FCT) techniques (see Chapters 9 and 10). LS methods propagate a linear function, which numerically is a much simpler problem. Regardless of the technique used, one finds that shear and vortical flowfields will tend to smooth and distort Φ. Fortunately, both TVD and FCT algorithms allow for limiters that keep the solution monotonic while enhancing the sharpness of the solution. For the TVD schemes Roe's Super-B limiter (Sweby (1984)) produces the desired effect. For FCT one increases the anti-diffusion by a small fraction (e.g. $c = 1.01$). The limiting procedure keeps the solution monotonic, while the increased anti-diffusion steepens Φ as much as is possible on a mesh. With these schemes, the discontinuity in Φ is captured within one to two gridpoints for all times. For LS the distance function Φ must be reinitialized periodically so that it truly represents the distance to the liquid–gas interface. The increase of CPU requirements can be kept to a minimum by using fast marching techniques and proper data structures (see Chapter 2, as well as Sethian (1999) and Osher and Fedkiw (2002)).

19.2.4. IMPOSITION OF CONSTANT MASS

Experience indicates that the amount of liquid mass (as measured by the region where the VOF indicator is larger than a cut-off value) does not remain constant for typical runs. The reasons for this loss or gain of mass are manifold: loss of steepness in the interface region, inexact divergence of the velocity field, boundary velocities, etc. This lack of exact conservation of liquid mass has been reported repeatedly in the literature (Sussman *et al.* (1994), Sussman and Puckett (2000), Enright *et al.* (2003)). The classic recourse is to add/remove mass in the interface region in order to obtain an exact conservation of mass. At the end of every timestep, the total amount of fluid mass is compared to the expected value. The expected value is determined from the mass at the previous timestep, plus the mass flux across all boundaries during the timestep. The differences in expected and actual mass are typically very small (less than 10^{-4}), so that quick convergence is achieved by simply adding and removing mass appropriately. The key question is *where* to add and remove mass. A commonly used approach is to make the mass taken/added proportional to the absolute value of the normal velocity of the interface:

$$v_n = \left| \mathbf{v} \cdot \frac{\nabla\Phi}{|\nabla\Phi|} \right|. \tag{19.21}$$

In this way the regions with no movement of the interface remain unaffected by the changes made to the interface in order to impose strict conservation of mass. The addition and removal of mass typically occurs at points close to the liquid–gas interface, where Φ does not assume extreme values. In some instances, the addition or removal of mass can lead to values of Φ outside the allowed range. If this occurs, the value is capped at the extreme value, and further corrections are carried out at the next iteration.

19.2.5. DEACTIVATION OF AIR REGION

Given that the air region is not treated/updated, any CPU spent on it may be considered wasted. Most of the work is spent in loops over the edges (upwind solvers, limiters, gradients, etc.). Given that edges have to be grouped in order to avoid memory contention/allow vectorization when forming RHSs (see Chapter 15), this opens a natural way of avoiding

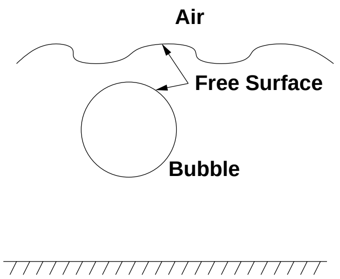


Figure 19.8. Bubble in water

unnecessary work: form relatively small edge groups that still allow for efficient vectorization, and deactivate groups instead of individual edges (see Chapter 16). In this way, the basic loops over edges do not require any changes. The `if`-test for whether an edge group is active or deactive occurs outside the inner loops over edges, leaving them unaffected. On scalar processors, edge groups as small as `negrp=8` may be used. Furthermore, if points and edges are grouped together in such a way that proximity in memory mirrors spatial proximity, most of the edges in air will not incur any CPU penalty.

19.2.6. TREATMENT OF BUBBLES

The treatment of bubbles follows the classic assumption that the timescales associated with the speed of sound in the bubble are much faster than the timescales of the surrounding fluid. This implies that at each instance the pressure in the bubble is (spatially) constant. As long as the bubble is not in contact with the atmospheric air (see Figure 19.8), the pressure can be obtained from the isentropic relation

$$\frac{p_b}{p_{b0}} = \left(\frac{\rho_b}{\rho_{b0}}\right)^{\gamma}, \tag{19.22}$$

where p_b, ρ_b denote the pressure and density in the bubble and p_{b0} and ρ_{b0} the reference values (e.g. those at the beginning of the simulation). The gas in the bubble is marked by solving a scalar advection equation of the form given by (19.20):

$$b_{,t} + \mathbf{v}_a \cdot \nabla b = 0, \tag{19.23}$$

where, initially, $b = 1.0$ denotes the bubble region and $b = 0.0$ the remainder of the flowfield. The same advection schemes and steepening algorithms as used for Φ are also used for b. At the beginning of every timestep the total volume occupied by the gas is added. From this volume the density is inferred, and the pressure is computed from (19.22).

At the end of every timestep, a check is performed to see whether the bubble has reached contact with the air. This happens if we have, at a given point, $b > 0.5$ and $\Phi > \Phi_{0.5}$. Should this be the case, the neighbour elements of these points that are in air are set to $b = 1.0$. This increases the volume occupied by the bubble, thereby reducing the pressure. Over the course of a few timesteps, the pressure in the bubble then reverts to atmospheric pressure, and one observes a rather quick bubble collapse.

19.2.7. ADAPTIVE REFINEMENT

As seen in Chapter 14, adaptive mesh refinement may be used to reduce CPU and memory requirements without compromising the accuracy of the numerical solution. For multiphase problems the mesh can be refined automatically close to the liquid–gas interface (Hay and Visonneau (2005), Löhner *et al.* (2006)). This may be done by including two additional refinement indicators (in addition to the usual ones based on the flow variables). The first one looks at the edges cut by the liquid–gas interface value of Φ, and refines the mesh to a certain element size or refinement level (Löhner and Baum (1992)). The second, more sophisticated indicator, looks at the liquid–gas interface curvature, given by

$$\kappa = \nabla \cdot \mathbf{n}, \qquad \mathbf{n} = \frac{\nabla\Phi}{|\nabla\Phi|}, \tag{19.24}$$

and refines the mesh only in regions where the element size is deemed insufficient.

19.2.8. EXAMPLES FOR SURFACE CAPTURING

19.2.8.1. Breaking dam problem

This is a classic test case for free surface flows. The problem definition is shown in Figure 19.9(a). This case was run on a coarse mesh with `nelem=16,562` elements, a fine mesh with `nelem=135,869` and an adaptively refined mesh (where the coarse mesh was the base mesh) with approximately `nelem=30,000` elements. The refinement indicator for the latter was the free surface (see above), and the mesh was adapted every five timesteps.

Figure 19.9(b) shows the discretization for the coarse mesh, and Figures 19.9(c)–(f) the development of the flowfield and the free surface until the column of water hits the right wall. Note the mesh adaptation in time. The results obtained for the horizontal location of the free surface along the bottom wall are compared to the experimental values of Martin and Moyce (1952), as well as the numerical results obtained by Hansbo (1992), Kölke (2005) and Walhorn (2002) in Figure 19.9(g). The dimensionless time and displacement are given by $\tau = t\sqrt{2g/a}$ and $\delta = x/a$, where a is the initial width of the water column. As one can see, the agreement is very good, even for the coarse mesh. The difference between the adaptively refined mesh and the fine mesh was almost indistinguishable, and therefore only the results for the fine mesh are shown in the graph.

19.2.8.2. Sloshing of a 2-D tank due to sway excitation

This example, taken from Löhner (2006), considers the sloshing of a partially filled 2-D tank. The main tank dimensions are $L = H = 1$ m, with tank width $B = 0.1$ m. The problem definition is shown in Figure 19.10(a). Experimental data for this tank with a filling level $h/L = 0.35$ have been provided by Olsen (1970), and reported in Faltisen (1974) and Olsen and Johnsen (1975), where the tank was undergoing a sway motion, i.e. the tank oscillates horizontally with law $x = A\sin(2\pi t/T)$. A wave gauge was placed 0.05 m from the right wall and the maximum wave elevation relative to a tank-fixed coordinate system was recorded. In the numerical simulations reported by Landrini *et al.* (2003) using the SPH method, the forced oscillation amplitude increases smoothly in time and reaches its steady regime value in 10 T.

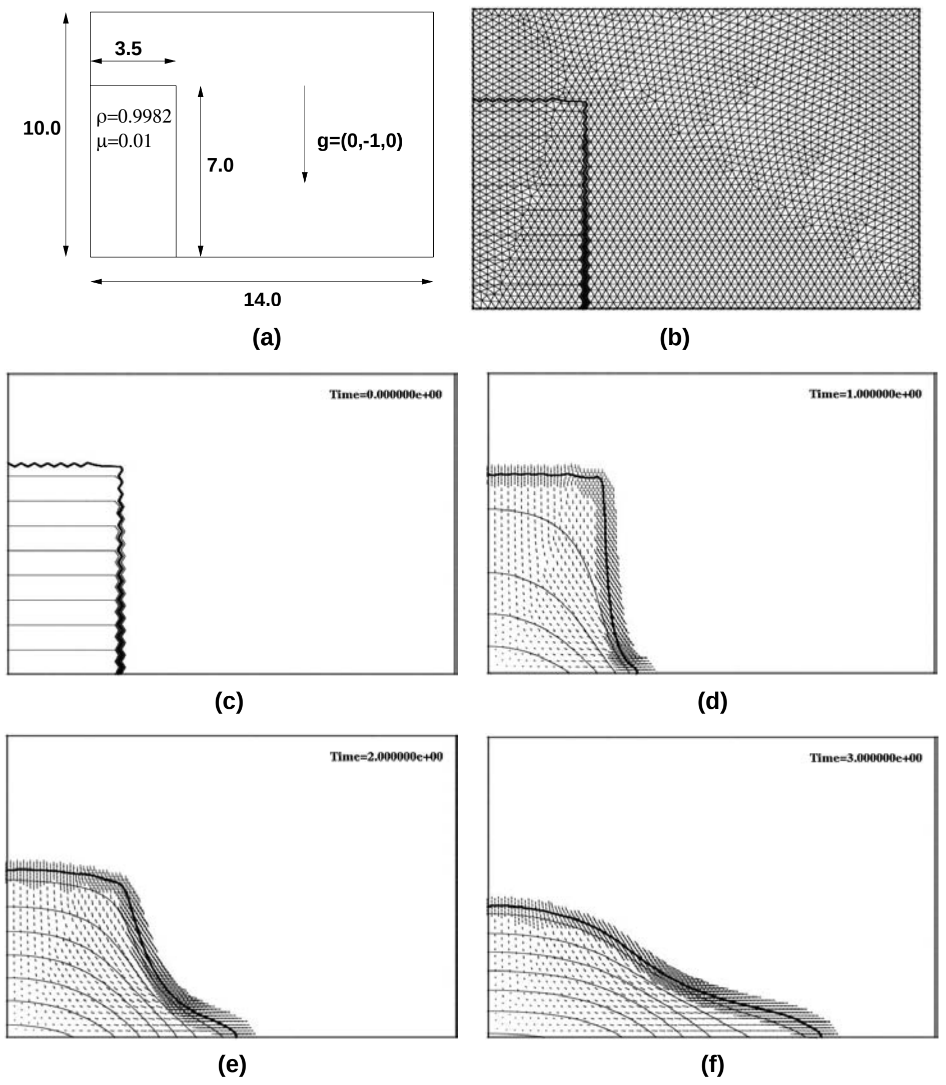
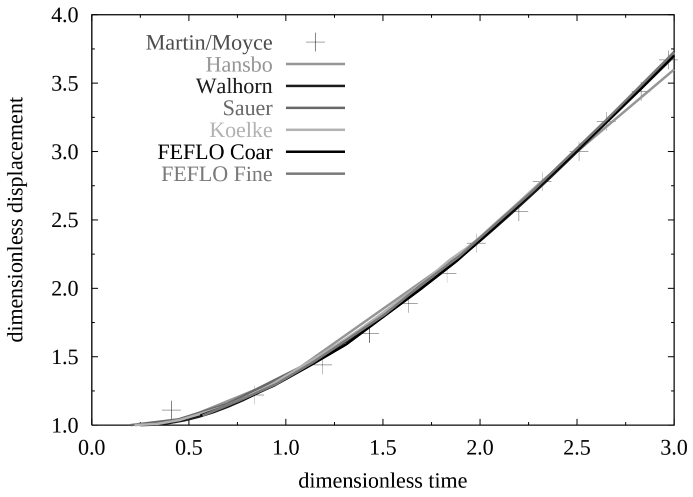


Figure 19.9. Breaking dam: (a) problem definition; (b) surface discretization for the coarse mesh; (c)–(f) flowfield at different times; (g) horizontal displacement

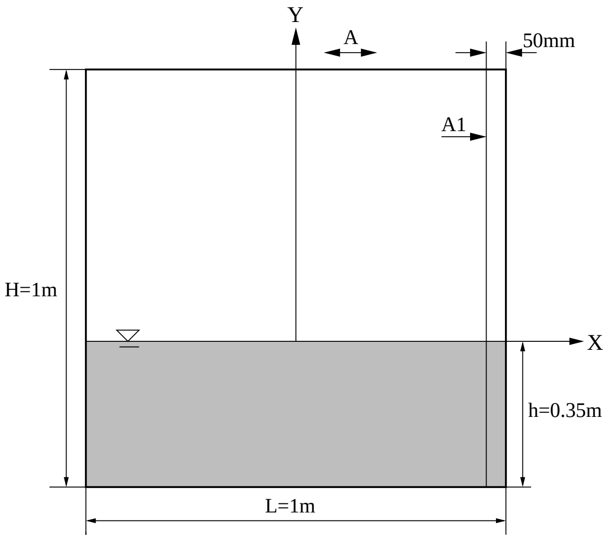
The simulation continues for another 30 T and the maximum wave elevation is recorded in the last 10 periods of oscillation.

The same procedure as in Landrini *et al.* (2003) was followed for the 32 cases computed. This corresponds to two amplitudes ($A = 0.025$, 0.05) and 16 periods, in the range $T = 1.0 - 1.8$ s or $T/T_1 = 0.787$–1.42, where $T_1 = 1.27$ s. When $h/L = 0.35$ the primary resonances of the first and the third modes occur at $T/T_1 = 1.0$ and $T/T_1 = 0.55$, respectively. The secondary resonance of the second mode is at $T/T_1 = 1.28$ (see Landrini *et al.* (2003)). The VOF results for the time history of the lateral force F_x when $T = 1.2, 1.3$ and



(g)

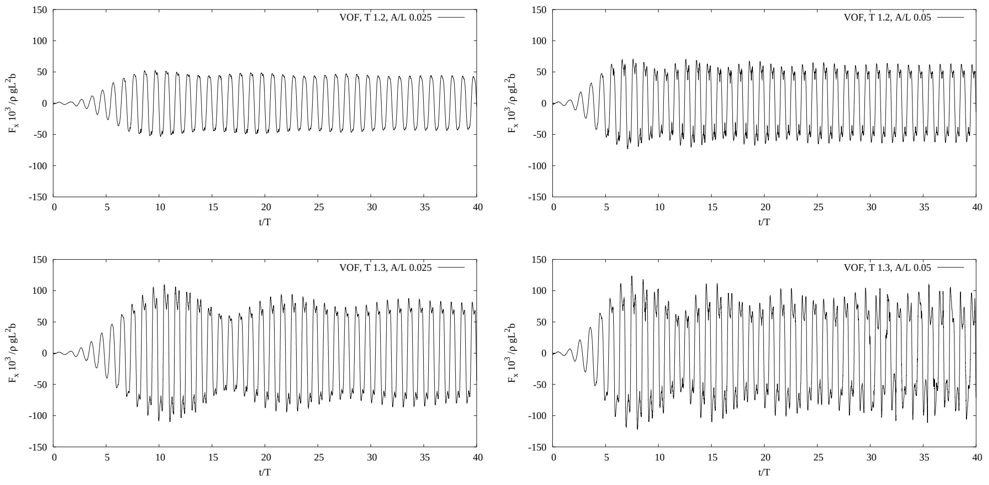
Figure 19.9. Continued



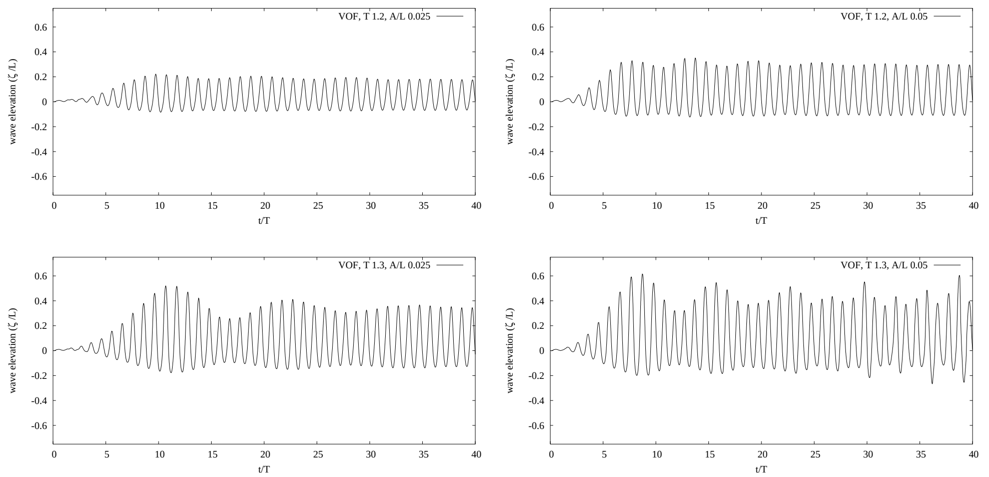
(a)

Figure 19.10. 2-D tank: (a) problem definition; (b) time history of lateral force F_x; (c) time history of wave elevation (probe A1); (d) snapshots of free surface wave elevation for $T = 1.3$ and $A/L = 0.05$; (e) maximum wave height (probe A1); (f), (g) maximum absolute values of lateral force F_x for $A/L = 0.025, 0.05$

$A = 0.025, 0.05$ are shown in Figure 19.10(b). The corresponding time history of the wave elevation at the wave probe A1 (see Figure 19.10(a)) are shown in Figure 19.10(c). Some free surface snapshots are shown in Figure 19.10(d). The dark line represents the free



(b)

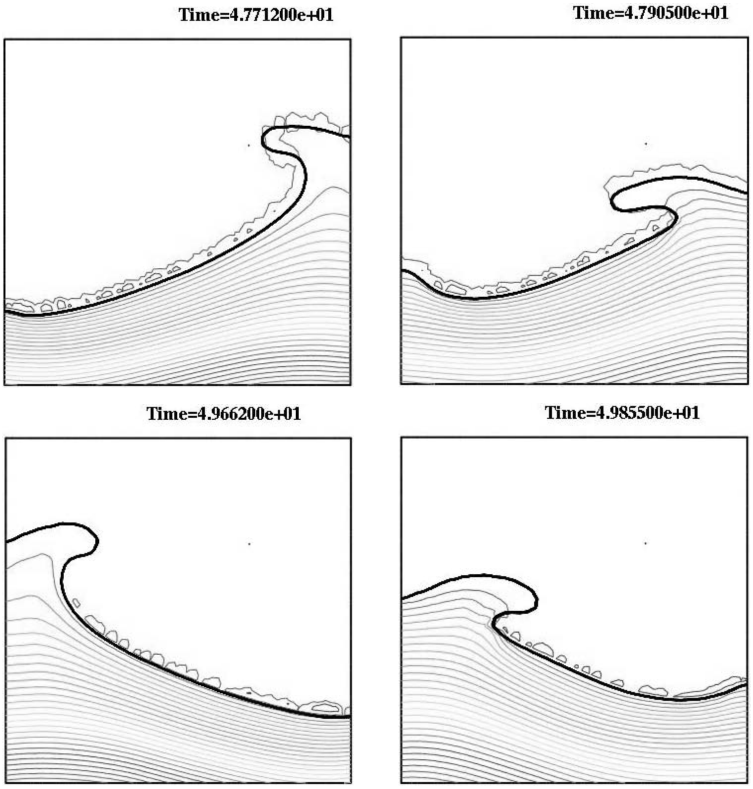


(c)

Figure 19.10. Continued

surface. Note also the 'undershoots' in the pressure due to extrapolation. The VOF results for maximum wave elevation ζ at the wave probe A1 (see Figure 19.10(a)) are compared with the experimental data and SPH results (Landrini *et al.* (2003)) in Figure 19.10(e) for $A/L = 0.025$, 0.05. Note that, as the wave inclination close to the wall is considerable, there is a non-negligible uncertainty in both the experiments and computational results.

The predicted lateral absolute values of maximum forces are compared with the experimental data and SPH results (Landrini *et al.* (2003)) in Figure 19.10(f) for $A/L = 0.05$. Figure 19.10(g) shows the comparison of predicted lateral absolute values of maximum forces for $A/L = 0.025$, 0.05. It can be seen from Figures 19.10(e)–(g) that both the maximum wave height and lateral absolute values of maximum forces predicted by the present VOF method



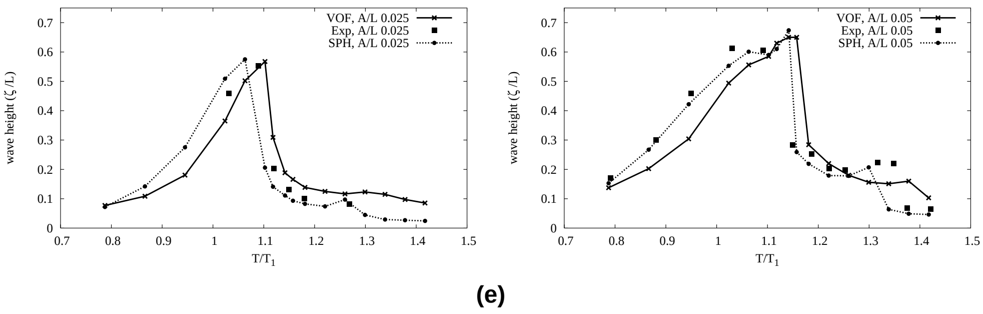
(d)

Figure 19.10. Continued

agree fairly well with the experimental data and SPH results, with a small phase shift among the three results. Figures 19.10(b) and (c) are typical time history plots. It should be noted from these figures that, even after a long simulation time (40 periods), steady-state results are not generally obtained. This is due to very small damping in the system. Landrini *et al.* (2003) noted the same behaviour in their numerical simulations. As a result, the predicted maximum wave elevation and the lateral absolute values of maximum forces plotted in Figure 19.10(e) are average maximum values for the last few periods for the cases when the steady state is not reached.

19.2.8.3. Sloshing of a 3-D tank due to sway excitation

In order to study the 3-D effects, the sloshing of a partially filled 3-D tank is considered. The main tank dimensions are $L = H = 1$ m, with tank width $b = 1$ m. The problem definition is shown in Figure 19.11(a). The 3-D tank has the same filling level $h/L = 0.35$ as the 2-D tank. The 3-D tank case is run on a mesh with `nelem=561,808` elements, and the 2-D tank is run on a mesh with `nelem=54,124` elements. The numerical simulations are carried out for both 3-D and 2-D tanks, where both tanks are undergoing the same prescribed sway motion



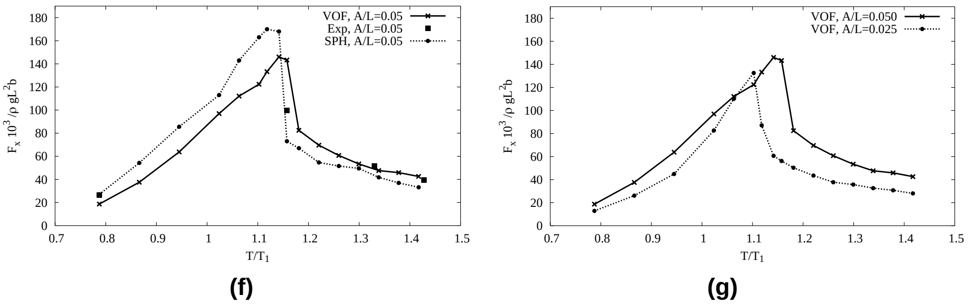
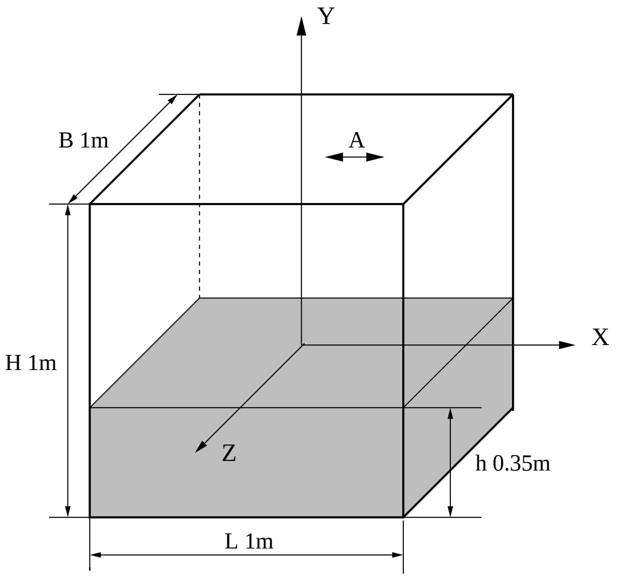
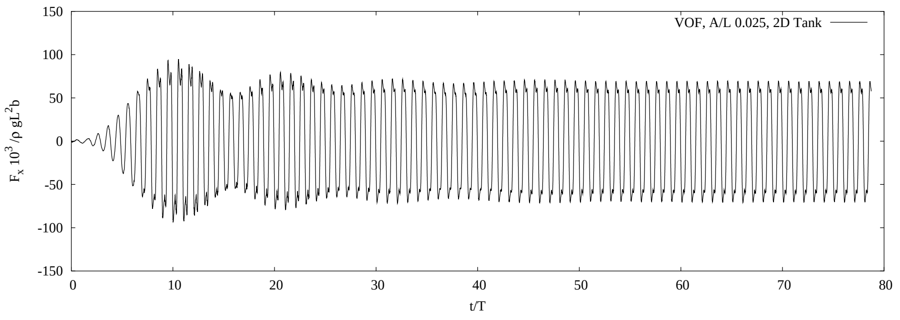


Figure 19.10. Continued

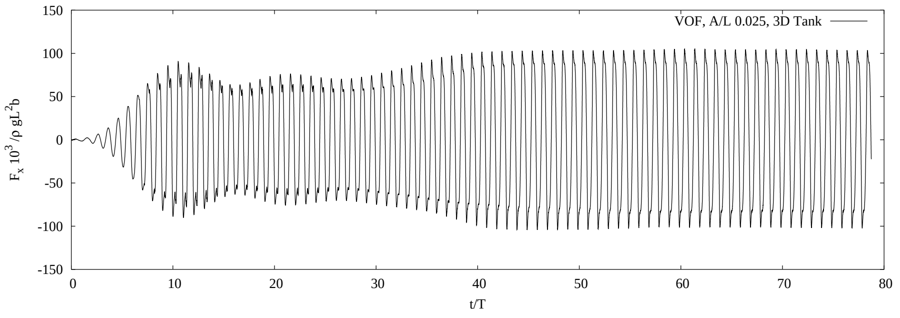


(a)

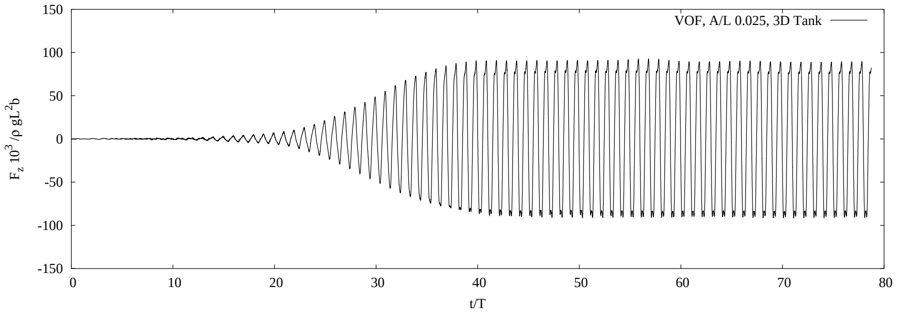
Figure 19.11. 3-D tank: (a) problem definition; time history of force F_x for (b) a 2-D tank and (c) a 3-D tank at $A/L = 0.025$, $T/T_1 = 1$; (d) time history of force F_z for a 3-D tank at $A/L = 0.025$, $T/T_1 = 1$; (e), (f), (g) snap shots of the free surface wave elevation for a 3-D tank



(b)



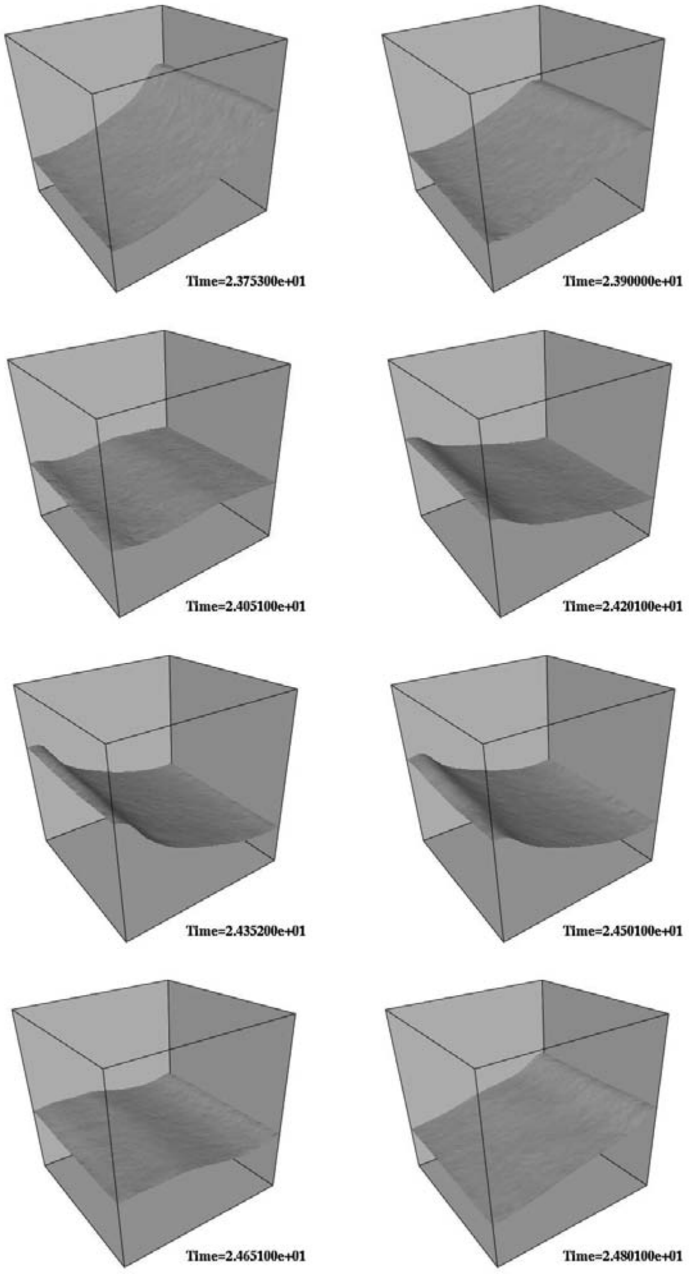
(c)



(d)

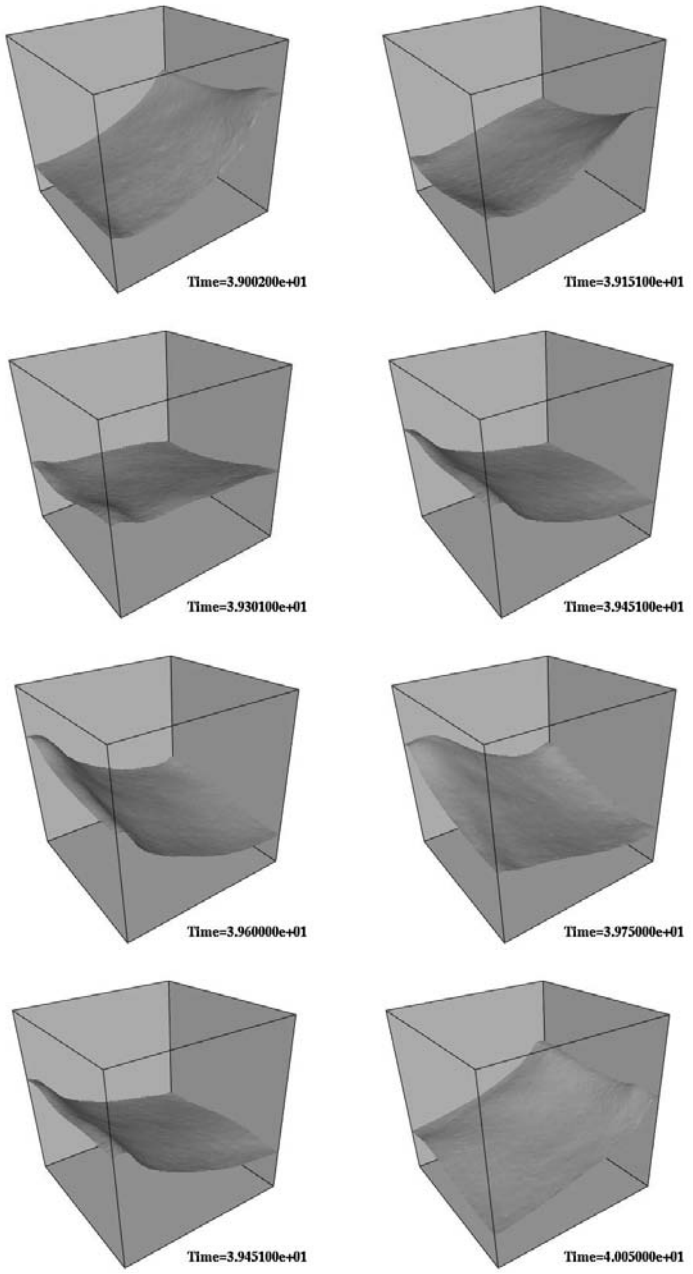
Figure 19.11. Continued

given by $x = A\ \sin(2\pi\ t/T)$. The simulations were carried out for $A = 0.025$ and $T = 1.27$ (i.e. $T/T_1 = 1$). The forced oscillation amplitude increases smoothly in time and reaches its steady regime value in 10 T. The simulation continues for another 70 T. In order to show the 3-D effects, the forces are non-dimensionalized with $\rho g L^2 b$ for both 2-D and 3-D tanks. Figures 19.11(b) and (c) show the time history of the force F_x (horizontal force in the same direction as the tank moving direction) for both 2-D and 3-D tanks. Figure 19.11(d) shows the time history of the force F_z (horizontal force perpendicular to the tank moving direction) for the 3-D tank. It is very interesting to observe from Figures 19.11(c) and (d) that there are



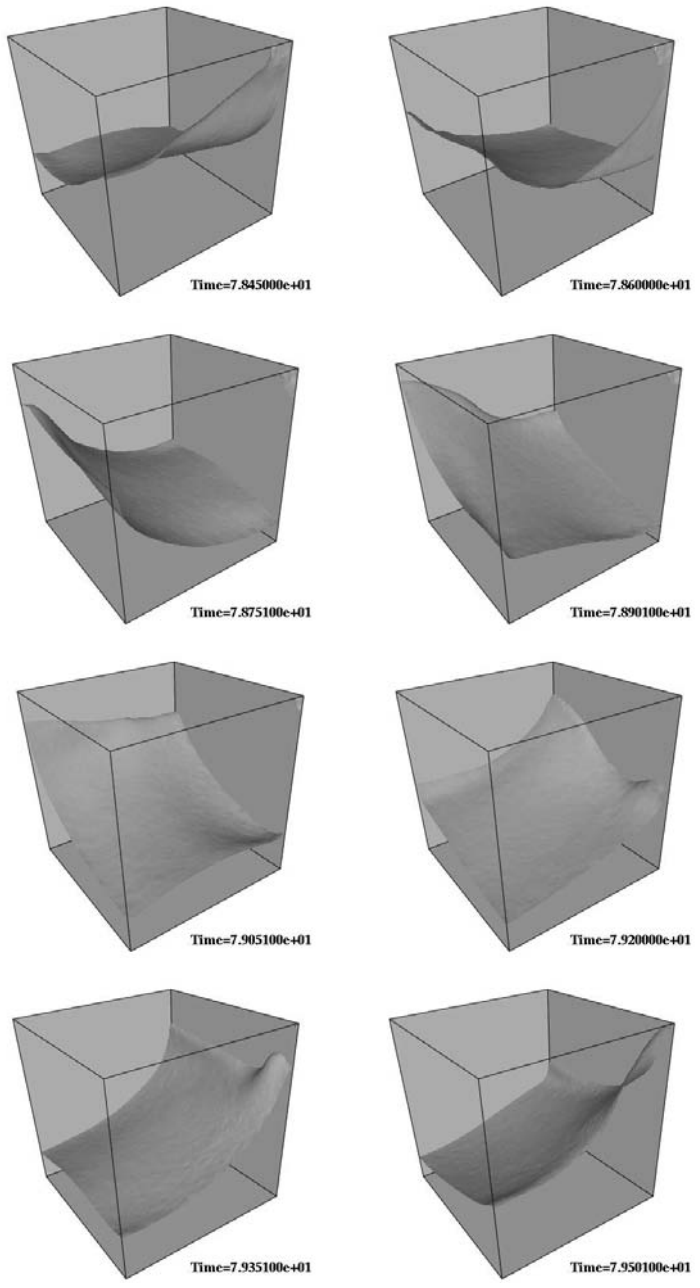
(e)

Figure 19.11. Continued



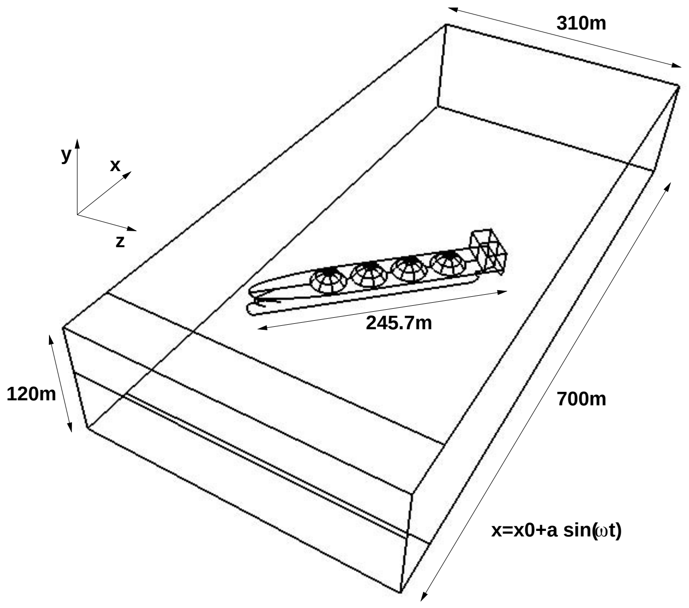
(f)

Figure 19.11. Continued

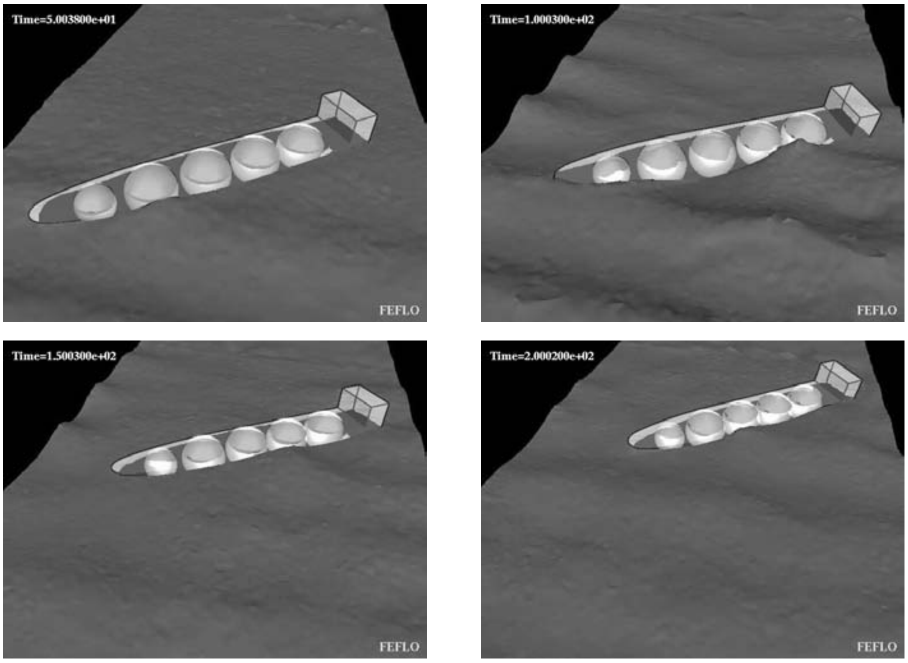


(g)

Figure 19.11. Continued

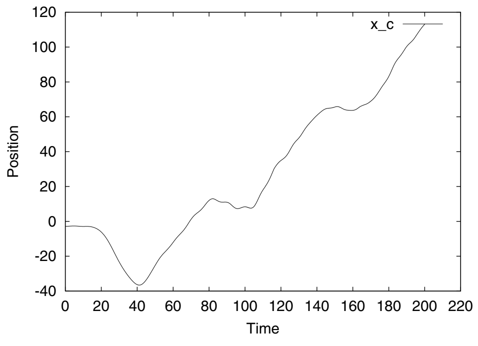


(a)

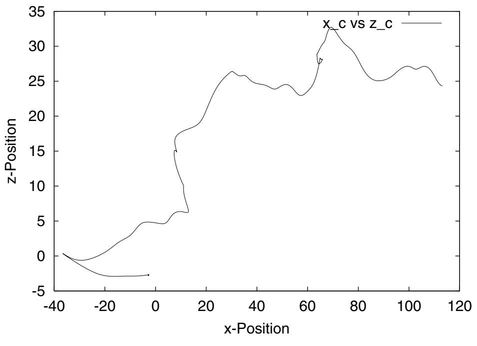


(b)

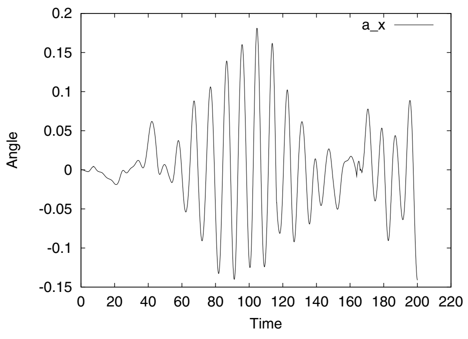
Figure 19.12. Ship adrift: (a) problem definition; (b) evolution of the free surface; (c), (d): position of center of mass; (e) roll angle versus time



(c)



(d)



(e)

Figure 19.12. Continued

almost no 3-D effects for the first 25 oscillating periods. The 3-D modes start to appear after 25 T, and fully build up at about 40 T. The 3-D flow pattern then remains steady and periodic for the rest of the simulation, which is about 40 more oscillation periods.

Figures 19.11(e)–(g) show a sequence of snapshots of the free surface wave elevation for the 3-D tank. For the first set of snapshots (see Figure 19.11(e)), the flow is still 2-D. The 3-D flow starts to build up in the second set of snapshots (see Figure 19.11(f)). The flow remains periodic 3-D for the last 40 periods. Figure 19.11(g) shows typical snapshots of the free surface for the last 40 periods. The 3-D effects are clearly shown in these plots.

19.2.8.4. Drifting ship

This example shows the use of interface capturing to predict the effects of drift in waves for large ships. The problem definition is given in Figure 19.12(a). The ship is a generic liquefied natural gas (LNG) tanker, and is considered rigid. The waves are generated by moving the left wall of the domain. A large element size was specified at the far end of the domain in order to dampen the waves. The mesh at the ‘wave-maker plane’ is moved using a sinusoidal excitation. The ship is treated as a free, floating object subject to the hydrodynamic forces of the water. The surface nodes of the ship move according to a 6-DOF integration of the rigid-body motion equations. Approximately 30 layers of elements close to the ‘wave-maker

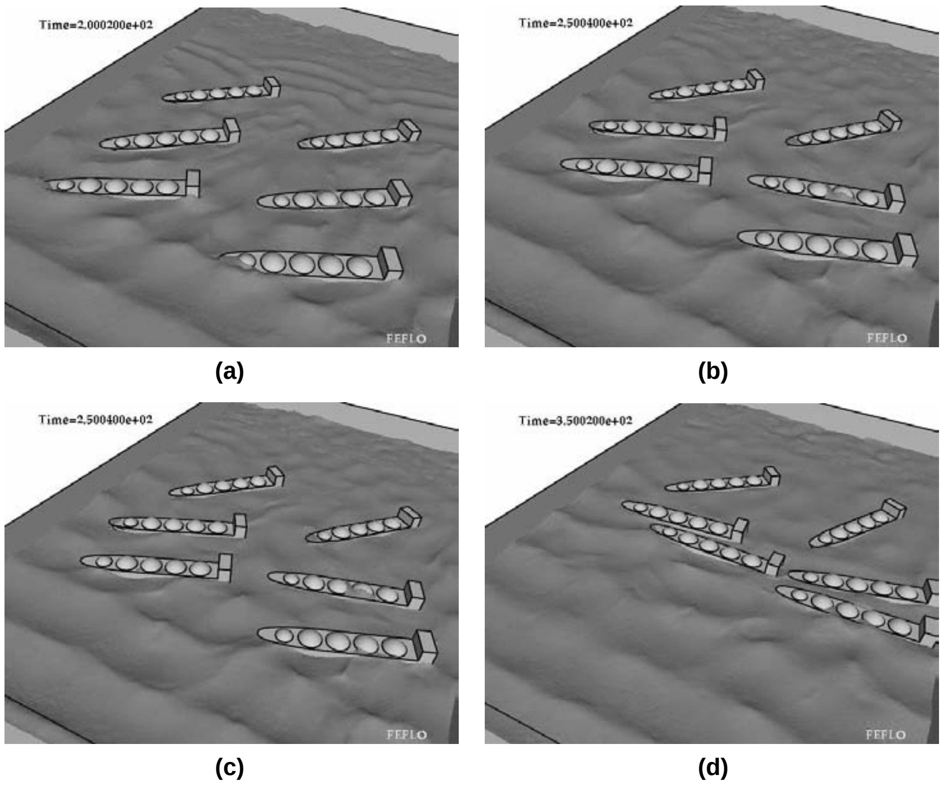


Figure 19.13. (a)–(d): LNG tanker fleet: evolution of the free surface

plane' and the ship are moved, and the Navier–Stokes/VOF equations are integrated using the arbitrary Lagrangian–Eulerian frame of reference. The LNG tanks are assumed to be 80% full. This leads to an interesting interaction of the sloshing inside the tanks and the drifting ship. The mesh had approximately `nelem=2,670,000` elements, and the integration to 3 minutes of real time took 20 hours on a PC (3.2 GHz Intel P4, 2 Gbytes RAM, Linux OS, Intel compiler). Figure 19.12(b) shows the evolution of the flowfield, and Figures 19.12(c) and (d) the body motion. Note the change in position for the ship, as well as the roll motion.

19.2.8.5. Drifting fleet of ships

This example shows the use of interface capturing to predict the effects of drift and shielding in waves for a group of ships. The ships are the same LNG tankers as used in the previous example, but the tanks are considered full. The boundary conditions and mesh size distribution are similar to the ones used in the previous example. The ships are treated as free, floating objects subject to the hydrodynamic forces of the water. The surface nodes of the ships move according to a 6-DOF integration of the rigid-body motion equations. Approximately 30 layers of elements close to the 'wave-maker plane' and the ships are moved, and the Navier–Stokes/VOF equations are integrated using the arbitrary Lagrangian–Eulerian frame of reference. The mesh had approximately 10 million elements and the integration to 6 minutes of real time took 10 hours on an SGI Altix using six processors (1.5 GHz Intel Itanium II, 8 Gbytes RAM, Linux OS, Intel compiler). Figures 19.13(a)–(d) show the evolution of the flowfield and the position of the ships. Note how the ships in the back are largely unaffected by the waves as they are ‘blocked’ by the ships in front, and how these ships cluster together due to wave forces.

19.2.9. PRACTICAL LIMITATIONS OF FREE SURFACE CAPTURING

Free surface capturing has been used to compute violent free surface flows with overturning waves and changes in topology. Even though in principle free surface capturing is able to compute all interface problems, some practical limitations do remain. The first and foremost is accuracy. For smooth surfaces, free surface fitting can yield far more accurate results with less gridpoints. This is even more pronounced for cases where a free surface boundary layer is present, as it is very difficult to generate anisotropic grids for the free surface capturing cases.

20 OPTIMAL SHAPE AND PROCESS DESIGN

The ability to compute flowfields implicitly implies the ability to optimize shapes and processes. The change of shape in order to obtain a desired or optimal performance is denoted as *optimal shape design*. Due to its immense industrial relevance, the relative maturity (accuracy, speed) of flow solvers and increasingly powerful computers, optimal shape design has elicited a large body of research and development (Newman *et al.* (1999), Mohammadi and Pironneau (2001)). The present chapter gives an introduction to the key ideas, as well as the optimal techniques to optimize shapes and processes.

20.1. The general optimization problem

In order to optimize a process or shape, a measurement of quality is required. This is given by one – or possibly many – so-called objective functions I, which are functions of design variables or input parameters $\boldsymbol{\beta}$, as well as field unknowns $\mathbf{u}$ (e.g. a flowfield)

$$I(\boldsymbol{\beta}, \mathbf{u}(\boldsymbol{\beta})) \rightarrow \min, \tag{20.1}$$

and is subject to a number of constraints.

- *PDE constraints*: these are the equations that describe the physics of the problem being considered, and may be written as

$$\mathbf{R}(\mathbf{u}) = 0. \tag{20.2}$$

- *Geometric constraints*:

$$\mathbf{g}(\boldsymbol{\beta}) \geq 0. \tag{20.3}$$

- *Physical constraints*:

$$\mathbf{h}(\mathbf{u}) \geq 0. \tag{20.4}$$

Examples for objective functions are:

- inviscid drag (e.g. for trans/supersonic airfoils): $I = \int_\Gamma pn_x \, d\Gamma$;
- prescribed pressure (e.g. for supercritical airfoils): $I = \int_\Gamma (p - p_0)^2 \, d\Gamma$;
- weight (e.g. for structures): $I = \int_\Omega \rho \, d\Omega$;
- uniformity of magnetic field (electrodynamics): $I = \int_\Omega (\mathbf{B} - \mathbf{B}_0)^2 \, d\Omega$.

Examples for PDE constraints $\mathbf{R}(\mathbf{u})$ are all the commonly used equations that describe the relevant physics of the problem:

- fluids: Euler/Navier–Stokes equations;
- structures: elasticity/plasticity equations;
- electromagnetics: Maxwell equations;
- etc.

Examples for geometric constraints $\mathbf{g}(\boldsymbol{\beta})$ are:

- wing area cross-section (stress, fuel): $A > A_0$;
- trailing edge thickness (cooling): $w > w_0$;
- width (manufacturability): $w > w_0$;
- etc.

Examples for physical constraints $\mathbf{h}(\mathbf{u})$ are:

- a constrained negative pressure gradient to avoid separation: $\mathbf{s} \cdot \nabla p > pg_0$;
- a constrained pressure to avoid cavitation: $p > p_0$;
- a constrained shear stress to avoid blood haemolysis: $|\tau| < \tau_0$;
- a constrained stress to avoid structural failure: $|\sigma| < \sigma_0$;
- etc.

Before proceeding, let us define with a higher degree of precision process and shape optimization. With shape optimization, we can clearly define three different optimization options (Jakiela *et al.* (2000), Kicinger *et al.* (2005)): topological optimization (TOOP), shape optimization (SHOP) and sizing optimization (SIOP). These options mirror the typical design cycle (Raymer (1999)): preliminary design, detailed design and final design. With reference to Figure 20.1, we can define the following.

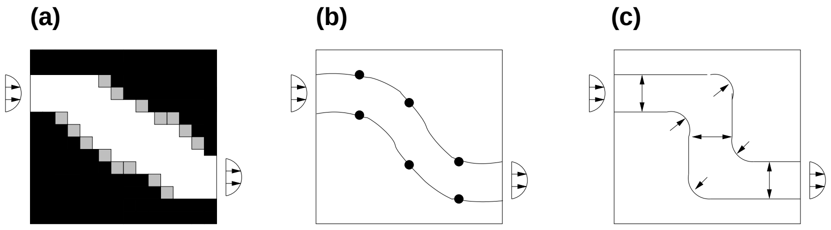


Figure 20.1. Different types of optimization: (a) topology; (b) shape; (c) sizing

- *Topological optimization*. The determination of an optimal material layout for an engineering system. TOOP has a considerable theoretical and empirical legacy in structural mechanics (Bendsoe and Kikuchi (1988), Jakiela *et al.* (2000), Kicinger *et al.* (2005), Bendsoe (2004)), where the removal of material from zones where low stress levels occur (i.e. no load bearing function is being realized) naturally leads to the common goal of weight minimization. For fluid dynamics, TOOP has been used for internal flow problems (Borrvall and Peterson (2003), Hassine *et al.* (2004), Moos *et al.* (2004), Guest and Prévost (2006), Othmer *et al.* (2006)).

- *Shape optimization*. The determination of an optimal contour, or shape, for an engineering system whose topology has been fixed. This is the classic optimization task for airfoil/wing design, and has been the subject of considerable research and development during the last two decades (Pironneau (1985), Jameson (1988, 1995), Kuruvila *et al.* (1995), Reuther and Jameson (1995), Reuther *et al.* (1996), Anderson and Venkatakrishnan (1997), Elliott and Peraire (1997, 1998), Mohammadi (1997), Nielsen and Anderson (1998), Medic *et al.* (1998), Reuther *et al.* (1999), Nielsen and Anderson (2001), Mohammadi and Pironneau (2001), Dreyer and Matinelli (2001), Soto and Löhner (2001a,b, 2002)).

- *Sizing optimization*. The determination of an optimal size distribution for an engineering system whose topology and shape has been fixed. A typical sizing optimization in fluid mechanics is the layout of piping systems for refineries. Here, the topology and shape of the pipes is considered fixed, and one is only interested in an optimal arrangement of the diameters.

For all these types of optimization (TOOP, SHOP, SIOP) the parameter space is defined by a set of variables $\boldsymbol{\beta}$. In order for any optimization procedure to be well defined, the set of design variables $\boldsymbol{\beta}$ must satisfy some basic conditions (Gen (1997)):

- *non-redundancy*: any process, shape or object can be obtained by a *one and only one* set of design variables $\boldsymbol{\beta}$;

- *legality*: any set of design variables $\boldsymbol{\beta}$ can be realized as a process, shape or object;

- *completeness*: any process, shape or object can be obtained by a set of design variables $\boldsymbol{\beta}$; this guarantees that any process, shape or object can be obtained via optimization;

- *causality* (continuity): small variations in $\boldsymbol{\beta}$ lead to small changes in the process, shape or object being optimized; this is an important requirement for the convergence of optimization techniques.

Admittedly, at first sight all of these conditions seem logical and easy to satisfy. However, it has often been the case that an over-reliance on ‘black-box’ optimization has led users to employ ill-defined sets of design variables.

20.2. Optimization techniques

Given the vast range of possible applications, as well as their immediate benefit, it is not surprising that a wide variety of optimization techniques have emerged. In the simplest case, the parameter space $\boldsymbol{\beta}$ is tested exhaustively. An immediate improvement is achieved by testing in detail only those regions were 'promising minima' have been detected. This can be done by emulating the evolution of life via 'survival of the fittest' criteria, leading to so-called genetic algorithms. With reference to Figure 20.2, for smooth functions I one can evaluate the gradient $I_{,\boldsymbol{\beta}}$ and change the design in the direction opposite to the gradient. In general, such gradient techniques will not be suitable to obtain globally optimal designs, but can be used to quickly obtain local minima. In the following, we consider in more detail the recursive exhaustive parameter scoping, genetic algorithms and gradient-based techniques. Here, we already note the rather surprising observation that with optimized gradient techniques and adjoint solvers the computational cost to obtain an optimal design is comparable to that of obtaining a single flowfield (!).

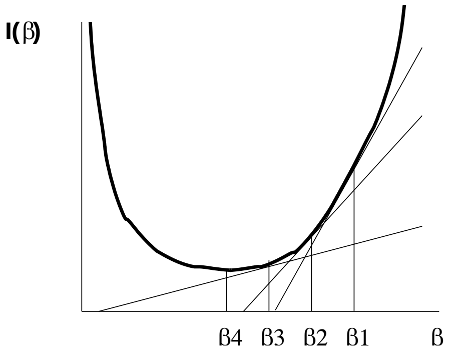


Figure 20.2. Local minimum via gradient-based optimization

20.2.1. RECURSIVE EXHAUSTIVE PARAMETER SCOPING

Suppose we are given the optimization problem

$$I(\boldsymbol{\beta}, \mathbf{u}(\boldsymbol{\beta})) \to \min. \tag{20.5}$$

In order to norm the design variables, we define a range $\beta^i_{\min} \leq \beta^i \leq \beta^i_{\max}$ for each design variable. An *instantiation* is then given by

$$\beta^i = (1 - \alpha^i)\beta^i_{\min} + \alpha^i \beta^i_{\max}, \tag{20.6}$$

implying $I(\boldsymbol{\beta}) = I(\boldsymbol{\beta}(\boldsymbol{\alpha}))$. By working only with the α^i, an abstract, non-dimensional, bounded ($[0, 1]$) setting is achieved, which allows for a large degree of commonality among various optimization algorithms.

The simplest (and most expensive) way to solve (20.1) is to divide each design parameter into regular intervals, evaluate the cost function for all possible combinations, and retain the best. Assuming n_d subdivisions per design variable and N design variables, this amounts to n_d^N cost function evaluations. Each one of these cost function evaluations corresponds to one (or several) CFD runs, making this technique suitable only for problems where N is relatively small. An immediate improvement is achieved by restricting the number of subdivisions n_d

to a manageable number, and then shrinking the parameter space recursively around the best design. While significantly faster, such a recursive procedure runs the risk of not finding the right minimum if the (unknown) local ‘wavelength’ of non-smooth functionals is smaller than the interval size chosen for the exhaustive search (see Figure 20.3).

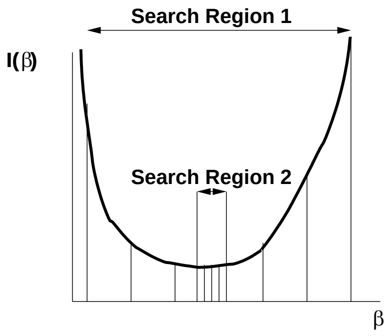


Figure 20.3. Recursive exhaustive parameter scoping

The basic steps required for the recursive exhaustive algorithm can be summarized as follows.

```
Ex1. Define:
       - Parameter space size for $\boldsymbol{\alpha}$ [0,1];
       - Nr. of intervals (interval length $h = 1/n_d$);
Ex2. while: $h > h_{\min}$:
Ex3. Evaluate the cost function $I(\boldsymbol{\beta}(\boldsymbol{\alpha}))$ for all possible combinations of $\alpha_i$;
Ex4. Retain the combination $\boldsymbol{\alpha}_{\text{opt}}$ with the lowest cost function;
Ex5. Define new search range: $[\boldsymbol{\alpha}_{\text{opt}} - h/2, \boldsymbol{\alpha}_{\text{opt}} + h/2]$
Ex6. Define new interval size: $h := h/n$
     end while
```

20.2.2. GENETIC ALGORITHMS

Given the optimization problem (20.1), a simple and very general way to proceed is by copying what nature has done in the course of evolution: try variations of $\boldsymbol{\beta}$ and keep the ones that minimize (i.e. improve) the cost function $I(\boldsymbol{\beta}, \mathbf{u}(\boldsymbol{\beta}))$. This class of optimization techniques are called *genetic algorithms* (Goldberg (1989), Deb (2001), De Jong (2006)) or *evolutionary algorithms* (Schwefel (1995)). The key elements of these techniques are:

- a *fitness measure*, given by $I(\boldsymbol{\beta}, \mathbf{u}(\boldsymbol{\beta}))$, to measure different designs against each other;
- *chromosome coding*, to parametrize the design space given by $\boldsymbol{\beta}$;
- *population size* required to achieve robust design;
- *selection*, to decide which members of the present/next generation are to be kept/used for reproductive purposes; and
- *mutation*, to obtain ‘offspring’ not present in the current population.

The most straightforward way to code the design variables into chromosomes is by defining them to be functions of the parameters $0 \leq \alpha^i \leq 1$. As before, an ***instantiation*** is given by

$$\beta^i = (1 - \alpha^i)\beta^i_{\min} + \alpha^i \beta^i_{\max}. \tag{20.7}$$

The population required for a robust selection needs to be sufficiently large. A typical choice for the number of individuals in the population M as compared to the number of chromosomes (design variables) N is

$$M > O(2N). \tag{20.8}$$

Given a population and a fitness measure associated with each individual, the next generation has to be determined. Depending on the life cycle and longevity of the species, as well as the climatic and environmental conditions, in nature several successful strategies have emerged. For many insect species, the whole population dies and is replaced when a new generation is formed. Denoting by μ the parent population and by λ the offspring population, this complete replacement strategy is written as (μ, λ). Larger mammals, as well as many birds, reptiles and fish, live long enough to produce several offspring at different times during their lifetime. In this case the offspring population consists of a part that is kept from the parent population, as well as new individuals that are fit enough to compete. This partial replacement strategy is written as $(\mu + \lambda)$. In order to achieve a monotonic improvement in designs, the $(\mu + \lambda)$ strategy is typically used, and a percentage of 'best individuals' of each generation is kept (typical value, $c_k = O(10\%)$). Furthermore, a percentage of 'worst individuals' are not admitted for reproductive purposes (typical value, $c_c = O(75\%)$). Each new individual is generated by selecting (randomly) a pair i, j from the allowed list of individuals and combining the chromosomes randomly. Of the many possible ways to combine chromosomes, we mention the following.

(a) *Chromosome splicing*. A random crossover point l is selected from the design parameters. The chromosomes for the new individual that fall below l are chosen from i, the rest from j:

$$\begin{aligned} \alpha_k &= \alpha^i_k, \quad 1 \leq k \leq l, \\ \alpha_k &= \alpha^j_k, \quad l \leq k \leq n. \end{aligned} \tag{20.9}$$

(b) *Arithmetic pairing*. A random pairing factor $-\xi < \gamma < 1 + \xi$ is selected and applied to all variables of the chromosomes in a uniform way. The chromosomes for the new individual are given by

$$\boldsymbol{\alpha} = (1 - \gamma)\boldsymbol{\alpha}^i + \gamma\boldsymbol{\alpha}^j. \tag{20.10}$$

Note that γ lies outside $[0, 1]$ (a typical value is $\xi = 0.2$). This is required, as otherwise the only way to reach locally outside the chromosome interval given by the pair i, j (or the present population) is via mutation, which is a slow and therefore expensive process.

(c) *Random pairing*. The arithmetic pairing can be randomized even further by choosing a different proportionality factor γ for each design variable. We then obtain

$$\alpha_k = (1 - \gamma_k)\alpha^i_k + \gamma_k\alpha^j_k. \tag{20.11}$$

Note that chromosome splicing and arithmetic pairing constitute particular cases of random pairing. The differences between these pairings can be visualized by considering the 2-D search space shown in Figure 20.4. If we have two points $\mathbf{x}_1, \mathbf{x}_2$ which are being paired to form a new point $\mathbf{x}_3$, then chromosome splicing, arithmetic pairing and random pairing lead to the regions shown in Figure 20.4. In particular, chromosome splicing only leads to two new possible point positions, arithmetic pairing to points along the line connecting $\mathbf{x}_1, \mathbf{x}_2$ and random pairing to points inside the extended bounding box given by $\mathbf{x}_1, \mathbf{x}_2$.

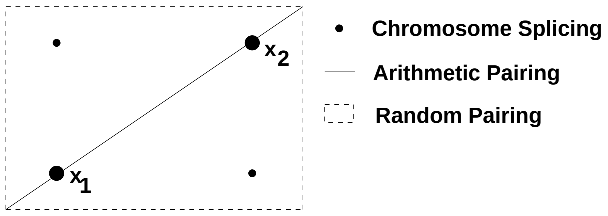


Figure 20.4. Regions for possible offspring from $\mathbf{x}_1, \mathbf{x}_2$

A population that is not modified continuously by mutations tends to become uniform, implying that the optimization may end in a local minimum. Therefore, a mutation frequency: $c_m = O(0.25/N)$ has to be applied to the new generation, modifying chromosomes randomly. The basic steps required per generation for genetic algorithms can be summarized as follows.

```
Ga1. Evaluate the fitness function $I(\boldsymbol{\beta}(\boldsymbol{\alpha}))$ for all individuals;
Ga2. Sort the population in ascending (descending) order of $I$;
Ga3. Retain the $c_k$ best individuals for the next generation;
Ga4. while: Population incomplete
     - Select randomly a pair $i, j$ from $c_c$ list
     - Obtain random pairing factors $0.0 < \gamma_k < 1.2$
     - Obtain the chromosomes for the new individual:
         $\boldsymbol{\alpha} = (1-\gamma)\,\boldsymbol{\alpha}^i + \gamma\,\boldsymbol{\alpha}^j$
     end while
```

For cases with a single, defined optimum, one observes that:

- the best candidate does not change over many generations – only the occasional mutation will yield an improvement, and thereafter the same pattern of unchanging best candidate will repeat itself;

- the top candidates (e.g. top 25% of population) become uniform, i.e. the genetic pool collapses.

Such a behaviour is easy to detect, and much faster convergence to the defined optimum can be achieved by 'randomizing' the population. If the chromosomes of any two individuals i, j are such that

$$d_{ij} = |\boldsymbol{\alpha}_i - \boldsymbol{\alpha}_j| < \epsilon, \tag{20.12}$$

the difference (distance) d_{ij} is artificially enlarged by adding/subtracting a random multiple of ϵ to one of the chromosomes. This process is repeated for all pairs i, j until none of them satisfies (20.12). As the optimum is reached, one again observes that the top candidate remains unchanged over many generations. The reason for this is that an improvement in the cost function can only be obtained with variations that are smaller than ϵ. When such a behaviour is detected, the solution is to reduce ϵ and continue. Typical reduction factors are 0.1–0.2. Given that $0 < \boldsymbol{\alpha} < 1$, a stopping criterion is automatically achieved for such cases: when the value of ϵ is smaller than a preset threshold, convergence has been achieved.

The advantages of genetic algorithms are manifold: they represent a completely general technique, able to go beyond local minima and hence are suitable for 'rough' cost functions I with multiple local minima. Genetic algorithms have been used on many occasions for shape optimization (see, e.g., Gage and Kroo (1993), Crispin (1994), Quagliarella and Cioppa (1994), Quagliarella (1995), Doorly (1995), Periaux (1995), Yamamoto and Inoue (1995), Vicini and Quagliarella (1997, 1999), Obayashi (1998), Obayashi *et al.* (1998), Zhu and Chan (1998), Naujoks *et al.* (2000), Pulliam *et al.* (2003)). On the other hand, the number of cost function evaluations (and hence field solutions) required is of $O(N^2)$, where N denotes the number of design parameters. The speed of convergence can also be strongly dependent on the crossover, mutation and selection criteria.

Given the large number of instantiations (i.e. detailed, expensive CFD runs) required by genetic algorithms, considerable efforts have been devoted to reduce this number as much as possible. Two main options are possible here:

- keep a database of *all generations/instantiations*, and avoid recalculation of regions already visited/ explored;

- replace the detailed CFD runs by approximate models.

Note that both of these options differ from the basic *modus operandi* of natural selection. The first case would imply selection from a semi-infinite population without regard to the finite life span of organisms. The second case replaces the actual organism by an approximate model of the same.

20.2.2.1. Tabu search

By keeping in a database the complete history of all individuals generated and evaluated so far, one is in a position to reject immediately offspring that:

- are too close to individuals already in the database;

- fall into regions populated by individuals whose fitness is low.

The regions identified as unpromising are labelled as 'forbidden' or 'tabu', hence the name.

20.2.2.2. Approximate models

As stated before, considerable efforts have been devoted to the development of approximate models. The key idea is to use these (cheaper) models to steer the genetic algorithm into the promising regions of the parameter space, and to use the expensive CFD runs as seldomly as possible (Quagliarella and Chinnici (2005)). The approximate models can be grouped into the following categories.

- *Reduced complexity models (RCMs)*. These replace the physical approximations in the expensive CFD run by simpler physical models. Examples of RCMs are potential solvers used as approximations to Euler/RANS solvers, or Euler/boundary-layer solvers used as approximations to RANS solvers.
- *Reduced degree of freedom models (RDOFMs)*. These keep the physical approximations of the expensive CFD run, but compute it on a mesh with far fewer degrees of freedom. Examples are all those approximate models that use coarse grid solutions to explore the design space.
- *General purpose approximators (GPAs)*. These use approximation theory to extrapolate the solutions obtained so far into unexplored regions of the parameter space. The most common of these are:
  - response surfaces, which fit a low-order polynomial through a vicinity of data points (Giannakoglov (2002));
  - neural networks, that are trained to reproduce the input–output obtained from the cost functions evaluated so far (Papila *et al.* (1999));
  - proper orthogonal decompositions (LeGresley and Alonso (2000));
  - kriging (Simpson *et al.* (1998), Kumano *et al.* (2006));
  - tessellations;
  - etc.

20.2.2.3. Constraint handling

The handling of constraints is remarkably simple for genetic algorithms. For any individual that does not satisfy the constraints given, one can either assign a very high cost function value (so that the individual is discarded in subsequent generations), or simply discard the individual directly while creating the next generation.

20.2.2.4. Pareto front

While an optimality criterion such as the one given by (20.1) forms a good basis for the derivation of optimization techniques, many design problems are defined by several objectives. One recourse is to modify (20.1) by writing the cost function as a sum of different criteria:

$$I(\boldsymbol{\beta}, \mathbf{u}(\boldsymbol{\beta})) = \sum_i c_i I_i(\boldsymbol{\beta}, \mathbf{u}(\boldsymbol{\beta})). \tag{20.13}$$

The important decision a designer or analyst has to make before starting an optimization is to select the criteria I_i *and* the weights c_i. Given that engineering is typically a compromise of different, conflicting criteria (styling, performance, cost, etc.), this step is not well defined. This implies that a proper choice of weights c_i before optimization may be difficult. Genetic algorithms can handle multiple design objectives concurrently using the concept of non-dominated individuals (Goldberg (1989), Deb (2001)). Given multiple objectives $I_i(\boldsymbol{\beta}, \mathbf{u}(\boldsymbol{\beta}))$, $i = 1, m$, the objective vector of individual k is partially less than the objective vector of individual k if

$$I_i(\boldsymbol{\beta}_k, \mathbf{u}(\boldsymbol{\beta}_k)) \leq I_i(\boldsymbol{\beta}_l, \mathbf{u}(\boldsymbol{\beta}_l)) \quad i = 1, m \quad \text{and} \quad \exists_j \backslash I_j(\boldsymbol{\beta}_k, \mathbf{u}(\boldsymbol{\beta}_k)) < I_j(\boldsymbol{\beta}_l, \mathbf{u}(\boldsymbol{\beta}_l)). \tag{20.14}$$

All individuals that satisfy these conditions with respect to all other individuals are said to be non-dominated. The key idea is to set the reproductive probabilities of all non-dominated individuals equally high. Therefore, before a new reproductive cycle starts, all individuals are ordered according to their optimality. In a series of passes, all non-dominated individuals are taken out of the population and assigned progressively lower probabilities and/or rankings (see Figure 20.5). The net effect of this sorting is a population that drifts towards the so-called Pareto front of optimal design. Additional safeguards in the form of niche formation and mating restrictions are required in order to prevent convergence to a single point (Deb (2001)). Note that one of the key advantages of such a Pareto ranking is that a multi-objective vector is reduced to a scalar: no weights are required, and the Pareto front gives a clear insight into the compromises that drive the design.

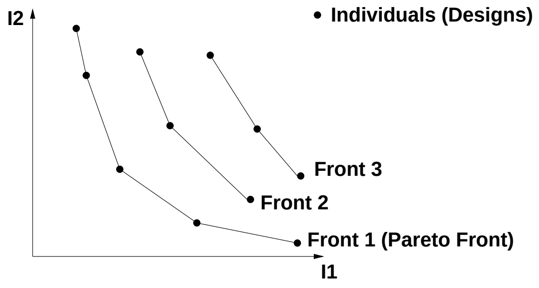


Figure 20.5. Pareto fronts for design problem with two objective functions

The visualization of Pareto fronts for higher numbers of criteria is the subject of current research (e.g. via data mining concepts (Obayashi (2002)).

20.2.3. GRADIENT-BASED ALGORITHMS

The second class of optimization techniques is based on evaluating *gradients* of $I(\boldsymbol{\beta}, \mathbf{u}(\boldsymbol{\beta}))$. From a Taylor series expansion we have

$$I + \Delta I \approx I + I_{,\boldsymbol{\beta}} \Delta \boldsymbol{\beta}. \tag{20.15}$$

This implies that if one chooses

$$\Delta \boldsymbol{\beta} = -\lambda I_{,\boldsymbol{\beta}}, \tag{20.16}$$

for sufficiently small λ the new functional has to diminish

$$I + \Delta I = I - \lambda I_{,\boldsymbol{\beta}}^T I_{,\boldsymbol{\beta}} \leq I. \tag{20.17}$$

The process has been sketched in Figure 20.2. As noted by Jameson (1995), a smoothed gradient can often be employed to speed up convergence. Denoting the gradient by $\mathbf{G} = I_{,\boldsymbol{\beta}}$, a simple Laplacian smoothing can be achieved via

$$\tilde{\mathbf{G}} - \nabla \mu \nabla \tilde{\mathbf{G}} = \mathbf{G}, \tag{20.18}$$

where $\tilde{\mathbf{G}}$ denotes the smoothed vector. Choosing

$$\Delta\boldsymbol{\beta} = -\lambda\tilde{\mathbf{G}} \tag{20.19}$$

leads to

$$\Delta I = -\mathbf{G}\lambda\tilde{\mathbf{G}} = -\lambda[\tilde{\mathbf{G}} - \nabla\mu\nabla\tilde{\mathbf{G}}]\tilde{\mathbf{G}}, \tag{20.20}$$

or, after integration by parts,

$$\Delta I = -\lambda[\tilde{\mathbf{G}}^2 + \mu(\nabla\tilde{\mathbf{G}})^2]. \tag{20.21}$$

This implies that a reduction in I is assured for arbitrary values of the smoothing parameter μ.

There exist a variety of ways to compute the required gradients $I_{,\boldsymbol{\beta}}$: automatic differentiation of flow codes, finite differences, response surfaces, etc. We elaborate on a few of these.

(a) *Automatic differentiation*. The complete chain of events required to obtain the objective function I for a set of design variables $\boldsymbol{\beta}$ includes:

- generation of CAD data S from design variables $\boldsymbol{\beta}$;
- generation of surface and volume mesh Ω from CAD data S;
- flow solution $CFD(\Omega)$;
- evaluation of cost function I from flow solution $CFD(\Omega)$.

Symbolically, these may be represented as follows:

$$\boldsymbol{\beta} \rightarrow S \rightarrow \Omega \rightarrow CFD \rightarrow I. \tag{20.22}$$

Each one of these stages represents several thousand lines of code. The key idea is to obtain all required derivatives directly from the original code (Griewank and Corliss (1991), Berz *et al.* (1996)). For example,

```
u = v*w  ⇒  du = dv*w + v*dw ,
u = v/w  ⇒  du = dv/w - v*dw/(w*w), etc.
```

Several efforts have been reported in this area, most notably the Automatic DIfferentiation of FORtran (ADIFOR) (Bischof *et al.* (1992), Hou *et al.* (1995)) and Odyssee (Rostaing *et al.* (1993)).

(b) *Finite differences*.

1. finite difference Perhaps the simplest way to compute gradients of I is via finite differences (Haftka (1985), Papay and Walters (1995), Hou *et al.* (1995), Besnard *et al.* (1998), Newman *et al.* (1999), Hino (1999), Miyata and Gotoda (2000), Tahara *et al.* (2000)). For each β_i, vary its value by a small amount $\Delta\beta_i$, recompute the cost function I and measure the gradient with respect to β_i:

$$I_{,\beta_i} = \frac{I(\boldsymbol{\beta} + \Delta\beta_i) - I(\boldsymbol{\beta})}{\Delta\beta_i}. \tag{20.23}$$

This implies $O(N)$ field solutions for each gradient evaluation. In order to achieve gradients of second order, β_i is varied in the positive and negative direction, requiring $O(2N)$ field solutions for each gradient evaluation. If gradients of second order are required, an interesting alternative is to use complex variables (Newman *et al.* (1998, 1999)). For these, we have

$$I(\boldsymbol{\beta} + i\Delta\beta_i) = I(\boldsymbol{\beta}) + i\Delta\beta_i I_{,\beta_i} + \text{hot},$$

implying

$$I_{,\beta_i} \approx \frac{\text{Im}[I(\boldsymbol{\beta} + i\Delta\beta_i)]}{\Delta\beta_i}.$$

Naturally, the question arises whether a single evaluation of the cost function using complex variables is faster than two evaluations using real variables. Anecdotal evidence indicates that this is indeed so, although some care is required when porting traditional CFD codes to complex variables. The most difficult (and troublesome) aspect of gradient evaluations via finite differences is the selection of a proper step size $\Delta\beta_i$ (Hou *et al.* (1995)). If the value selected is too low, the change in geometry is imperceptible to the solver (after all, mesh size is finite), and the result may be incorrect due to noise. If the value of $\Delta\beta_i$ is too high, the change in geometry may be sufficient to trigger a considerable change in the physics of the problem (e.g. a new separation zone occurs), leading to incorrect gradients.

(c) *Response surface*. An alternative to direct finite difference evaluation, which can be ‘noisy’ and hence inaccurate, is to populate the parameter space in a region close to the present design, and to fit a low-order polynomial through these data points. This technique, called *response surface*, also requires $O(N)$ field solutions for each gradient evaluation, and has seen widespread use (Narducci *et al.* (1995), Minami *et al.* (2002)). There are four major steps involved in constructing a response surface:

- selection of function type for the response surface;
- selection of the number of objective function evaluations to be used as the database to fit the response surface;
- determination of the location for the points used for fitting the response surface; and
- solution of the least-squares problem to fit the function.

As the design is optimized, the extent of the region where the surface is fitted is reduced.

20.2.3.1. Steepest descent

Given:

- the current objective function $I(\boldsymbol{\beta}(\boldsymbol{\alpha}_0))$,
- the gradient $I_{,\boldsymbol{\alpha}_0}$ and
- a first guess for the stepsize λ,

the design variables are changed according to

$$\Delta\boldsymbol{\alpha} = -\lambda I_{,\boldsymbol{\alpha}_0}. \tag{20.24}$$

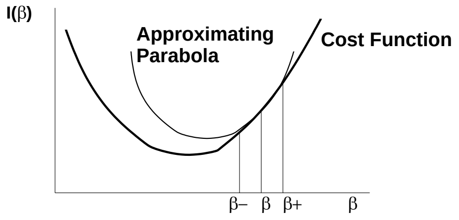


Figure 20.6. Parabolic extrapolation for step estimation

One can attempt to obtain a better guess for the step size λ by using a Taylor series expansion (which is akin to a quadratic extrapolation, see Figure 20.6). The aim is to obtain an increment $\Delta\boldsymbol{\alpha}$ such that the gradient $I_{,\boldsymbol{\alpha}}$ vanishes. With

$$I_{,\boldsymbol{\alpha}}(\boldsymbol{\alpha}_0 + \Delta\boldsymbol{\alpha}) = I_{,\boldsymbol{\alpha}}(\boldsymbol{\alpha}_0) + I_{,\boldsymbol{\alpha}\boldsymbol{\alpha}}(\boldsymbol{\alpha}_0)\Delta\boldsymbol{\alpha} = 0 \tag{20.25}$$

we have

$$\Delta\boldsymbol{\alpha} = -\frac{I_{,\boldsymbol{\alpha}}(\boldsymbol{\alpha}_0)}{I_{,\boldsymbol{\alpha}\boldsymbol{\alpha}}(\boldsymbol{\alpha}_0)}, \tag{20.26}$$

or, using finite differences along the gradient direction with step size $\Delta\mathbf{u}$,

$$\lambda_{\min} = \Delta\boldsymbol{\alpha} = -\frac{(I_1 - I_{-1})}{2(I_1 - 2I_0 + I_{-1})}\Delta\mathbf{u}. \tag{20.27}$$

Care has to be taken not to move the solution in the opposite direction. This can happen when one falsely senses a local maximum, a condition that can be avoided by imposing

$$\Delta\boldsymbol{\alpha} \cdot I_{,\boldsymbol{\alpha}_0} < 0. \tag{20.28}$$

The basic steepest descent step may be summarized as follows:

```
Sd1.  Guess Small $\lambda$
Sd2.  Evaluate $I_1,\ I_{-1}$
Sd3.  Obtain $\lambda_{\min}$
Sd4.  $\boldsymbol{\alpha} = \boldsymbol{\alpha} - \lambda_{\min} I_{,\boldsymbol{\alpha}_0}$
Sd5.  Evaluate $I_2 = I(\boldsymbol{\beta}(\boldsymbol{\alpha}))$
Sd6.  If: $I_2 > I_0 \Rightarrow$ Reduce $\lambda$ (goto Sd1)
Sd7.  Set: $\lambda = \lambda_{\min}$
Sd8.  do: icont=1,mcont ! Continuation Steps
      - $\boldsymbol{\alpha}_3 = \boldsymbol{\alpha} - \lambda I_{,\boldsymbol{\alpha}_0}$
      - Evaluate $I_3 = I(\boldsymbol{\beta}(\boldsymbol{\alpha}_3))$
      - If: $I_3 > I_2$: exit loop
      - Replace $I_2 = I_3,\ \boldsymbol{\alpha} = \boldsymbol{\alpha}_3$
      enddo
```

The convergence behaviour of genetic and gradient-based techniques can be illustrated by considering the following very simple problem: approximate $\sin(\omega x)$, $0 \le x \le 1$ via point collocation at regular intervals. The cost function is given by

$$I = \frac{1}{N}\sqrt{\sum_N (a_i - \sin(\omega x_i)^2)}, \tag{20.29}$$

where N denotes the number of collocation points (intervals) and a_i are the approximation parameters. Figure 20.7 shows the convergence history and the number of required function evaluations for the different techniques. One can see that the number of steps required for the gradient-based technique (based here on finite differences) is independent of the number of optimization variables N. On the other hand, the number of steps required for the genetic algorithm grows roughly linearly with the number of design variables.

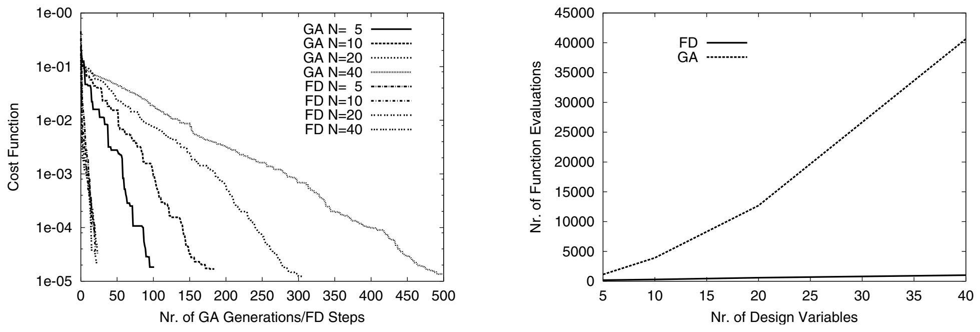


Figure 20.7. Convergence history and function evaluations

20.3. Adjoint solvers

All of the gradient techniques described so far have required of the order of one flowfield evaluation per design variable. This may be viable if the shape can be optimized with only a few design variables, a condition seldomly met in practice. A remarkable decrease of work may be achieved by using so-called adjoint variables (Pironneau (1973, 1974, 1985), Jameson (1988, 1995), Kuruvila *et al.* (1995), Anderson and Venkatakrishnan (1997), Elliott and Peraire (1997, 1998), Mohammadi (1997), Nielsen and Anderson (1998), Medic *et al.* (1998), Reuther *et al.* (1999), Mohammadi and Pironneau (2001), Dreyer and Matinelli (2001), Soto and Löhner (2001, 2002)). Consider a variation in the objective function I *and* the PDE constraint $\mathbf{R}$:

$$\delta I = I_{,\boldsymbol{\beta}}\delta\boldsymbol{\beta} + I_{,\mathbf{u}}\delta\mathbf{u}, \tag{20.30}$$

$$\delta \mathbf{R} = \mathbf{R}_{,\boldsymbol{\beta}}\delta\boldsymbol{\beta} + \mathbf{R}_{,\mathbf{u}}\delta\mathbf{u} = 0. \tag{20.31}$$

One may now introduce a Lagrange multiplier $\boldsymbol{\Psi}$ to merge these two expressions:

$$\delta I = I_{,\boldsymbol{\beta}}\delta\boldsymbol{\beta} + I_{,\mathbf{u}}\delta\mathbf{u} + \boldsymbol{\Psi}^T[\mathbf{R}_{,\boldsymbol{\beta}}\delta\boldsymbol{\beta} + \mathbf{R}_{,\mathbf{u}}\delta\mathbf{u}]. \tag{20.32}$$

After rearrangement of terms this results in

$$\delta I = [I_{,\boldsymbol{\beta}} + \boldsymbol{\Psi}^T \mathbf{R}_{,\boldsymbol{\beta}}]\delta\boldsymbol{\beta} + [I_{,\mathbf{u}} + \boldsymbol{\Psi}^T \mathbf{R}_{,\mathbf{u}}]\delta\mathbf{u}. \tag{20.33}$$

This implies that if one can solve

$$\boldsymbol{\Psi}^T \mathbf{R}_{,\mathbf{u}} = -I_{,\mathbf{u}}, \tag{20.34}$$

the variation of I is given by

$$\delta I = [I_{,\boldsymbol{\beta}} + \boldsymbol{\Psi}^T \mathbf{R}_{,\boldsymbol{\beta}}]\delta\boldsymbol{\beta} = [G^I]^T \delta\boldsymbol{\beta}. \tag{20.35}$$

The consequences of this rearrangement are profound:

- the variation of I exhibits only derivatives with respect to $\boldsymbol{\beta}$, i.e. no explicit derivatives with respect to **u** appear;
- the cost for the evaluation of gradients is *independent of the number of design variables* (!).

A design cycle using the adjoint approach is then composed of the following steps (see Figure 20.8 and (20.22)):

- with current design variables $\boldsymbol{\beta}$:
  - generate surface $\rightarrow S$ and volume $\rightarrow \Omega$ mesh, or
  - move mesh;
- with current mesh, solve flow field $\rightarrow$ **u**;
- with current mesh and **u**, solve adjoint field $\rightarrow \boldsymbol{\Psi}$;
- with current mesh and **u**, $\boldsymbol{\Psi}$, obtain gradients $\rightarrow I_{,\boldsymbol{\beta}}$;
- update design variables $\rightarrow \boldsymbol{\beta}$.

In the following, we consider in more detail each of the ingredients required. The original equation for the adjoint is given by

$$\boldsymbol{\Psi}^T \mathbf{R}_{,\mathbf{u}} = I^{\Omega}_{,\mathbf{u}} + I^{\Gamma}_{,\mathbf{u}}; \quad \int_{\Omega} \boldsymbol{\Psi}^T \mathbf{R}_{,\mathbf{u}}\, d\Omega = -\int_{\Gamma} I_{,\mathbf{u}}\, d\Gamma, \tag{20.36}$$

where I^{Ω} denotes a cost function defined over the domain, and I^{Γ} a cost function defined over the boundary. Note that, for most shape optimization problems, $I^{\Omega} = 0$.

20.3.1. ADJOINT EQUATIONS: RESIDUALS WITH FIRST DERIVATIVES AND SOURCE TERMS

If **R** can be written as a system of conservation laws of the form

$$\mathbf{R} = \mathbf{F}^i_{,i} - \mathbf{s} = 0, \tag{20.37}$$

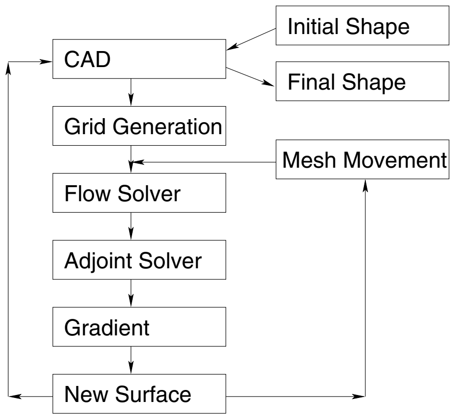


Figure 20.8. Design cycle with adjoint variables

where $\mathbf{F}(\mathbf{u})$ denotes a flux function and $\mathbf{s}$ a source term, to first order one can exchange the partial derivatives,

$$\mathbf{\Psi}^T \mathbf{R}_{,\mathbf{u}} = \mathbf{\Psi}^T (\mathbf{F}^i_{,i} - \mathbf{s})_{,\mathbf{u}} = \mathbf{\Psi}^T [(\mathbf{F}^i_{,\mathbf{u}})_{,i} - \mathbf{s}_{,\mathbf{u}}] = \mathbf{\Psi}^T [\mathbf{A}^i_{,i} - \mathbf{C}] = -I_{,\mathbf{u}}, \tag{20.38}$$

where $\mathbf{A}^i = \mathbf{F}^i_{,\mathbf{u}}$ denotes the flux Jacobian and $\mathbf{C} = \mathbf{s}_{,\mathbf{u}}$ the source Jacobian. Integration by parts yields the PDE for the adjoint:

$$-[\mathbf{A}^i]^T \mathbf{\Psi}_{,i} - \mathbf{C}^T \mathbf{\Psi} = -I^{\Omega}_{,\mathbf{u}}, \tag{20.39}$$

with boundary conditions

$$\int_\Gamma [\mathbf{A}^i n_i]^T \mathbf{\Psi}\, d\Gamma = -I^{\Gamma}_{,\mathbf{u}}. \tag{20.40}$$

Note that:

(a) a *linear* system of *advection equations* is obtained;

(b) the eigenvalues of this system are the same as those of the original PDE;

(c) due to the negative sign in front of the Jacobians $\mathbf{A}^i$, the 'advection direction' for the adjoint is opposite to the advection direction of the original PDE ($\mathbf{R}$).

20.3.2. ADJOINT EQUATIONS: RESIDUALS WITH SECOND DERIVATIVES

Suppose $\mathbf{R}$ can be written as a 'Laplacian' of the form

$$\mathbf{R} = -\nabla\mu\nabla\mathbf{w} = 0, \tag{20.41}$$

where μ denotes a diffusion or viscosity, and $\mathbf{w}(\mathbf{u})$ a set of different (e.g. non-conserved) variables. To first order one may again exchange the partial derivatives,

$$\mathbf{\Psi}^T \mathbf{R}_{,\mathbf{u}} = -\mathbf{\Psi}^T (\nabla\mu\nabla\mathbf{w})_{,\mathbf{u}} = -\mathbf{\Psi}^T \nabla\mu\nabla\mathbf{w}_{,\mathbf{u}} = -\mathbf{\Psi}^T \nabla\mu\nabla\mathbf{B} = -I_{,\mathbf{u}}, \tag{20.42}$$

where $\mathbf{B} = \mathbf{w}_{,\mathbf{u}}$ denotes the Jacobian of $\mathbf{w}$. Repeated integration by parts yields the PDE for the adjoint:

$$-\mathbf{B}^T \nabla\mu\nabla\boldsymbol{\Psi} = -I^{\Omega}_{,\mathbf{u}}, \tag{20.43}$$

with boundary conditions

$$-\int_\Gamma \mathbf{B}^T_{,n}\mu\boldsymbol{\Psi}\, d\Gamma + \int_\Gamma \mathbf{B}^T \mu\boldsymbol{\Psi}_{,n}\, d\Gamma = -I^{\Gamma}_{,\mathbf{u}}. \tag{20.44}$$

Note that:

(a) as with first-order derivatives, a *linear* system of *diffusion equations* is obtained;

(b) the sign of the operator has not changed, i.e. diffusion of the adjoint variables occurs in the same way as with the original PDE variables; this was to be expected as the diffusion (Laplacian) operator is self-adjoint.

In most applications, the residual (or PDE describing the field solution) will contain a combination of first and second spatial derivatives, as well as source terms (no spatial derivatives). As these are additive, the same sum applies to the adjoints.

20.3.3. JACOBIANS FOR EULER/NAVIER–STOKES EQUATIONS

For the sake of completeness, we include the relevant Jacobians for the compressible and incompressible Euler/Navier–Stokes equations.

20.3.3.1. Compressible Euler equations

The compressible Euler equations are given by

$$\mathbf{u}_{,t} + \nabla\cdot\mathbf{F} = 0, \tag{20.45}$$

where

$$\mathbf{u} = \begin{Bmatrix} \rho \\ \rho v_i \\ \rho e \end{Bmatrix}, \quad \mathbf{F}^j = \begin{Bmatrix} \rho v_j \\ \rho v_i v_j + p\delta_{ij} \\ v_j(\rho e + p) \end{Bmatrix}. \tag{20.46}$$

Here ρ, p, e and v_i denote the density, pressure, specific total energy and fluid velocity in direction x_i, respectively. This set of equations is closed by providing an equation of state for the pressure, e.g. for a polytropic gas,

$$p = (\gamma - 1)\rho[e - \tfrac{1}{2}v_j v_j], \tag{20.47}$$

where γ is the ratio of specific heats. Denoting $u = v_1$, $v = v_2$, $w = v_3$, $q = u^2 + v^2 + w^2$ and $c_1 = \gamma - 1$, $c_2 = c_1/2$, $c_3 = 3 - \gamma$, the Jacobian matrices are given by

$$\mathbf{A}^x = \begin{Bmatrix} 0; & 1; & 0; & 0; & 0; \\ -u^2 + c_2 q; & c_3 u; & -c_1 v; & -c_1 w; & c_1; \\ -uv; & v; & u; & 0; & 0; \\ -uw; & w; & o; & u; & 0; \\ -(\gamma e - c_1 q)u; & \gamma e - c_2 q - c_1 u^2; & -c_1 uv; & -c_1 uw; & \gamma u; \end{Bmatrix},$$

$$\mathbf{A}^y = \begin{Bmatrix} 0; & 0; & 1; & 0; & 0; \\ -uv; & v; & u; & 0; & 0; \\ -v^2 + c_2 q; & -c_1 uv; & c_3 v; & -c_1 w; & c_1; \\ -uw; & 0; & w; & v; & 0; \\ -(\gamma e - c_1 q)v; & -c_1 uv; & \gamma e - c_2 q - c_1 v^2; & -c_1 vw & \gamma v; \end{Bmatrix},$$

$$\mathbf{A}^z = \begin{Bmatrix} 0; & 0; & 0; & 1; & 0; \\ -uv; & w; & 0; & u; & 0; \\ -uw; & 0; & w; & v; & 0; \\ -w^2 + c_2 q; & -c_1 u; & -c_1 v; & c_3 w; & c_1; \\ -(\gamma e - c_1 q)w; & -c_1 uw; & -c_1 uw; & \gamma e - c_2 q - c_1 w^2; & \gamma w; \end{Bmatrix}. \quad (20.48)$$

20.3.3.2. *Incompressible Euler/Navier–Stokes equations*

The incompressible Euler/Navier–Stokes equations are given by

$$\mathbf{u}_{,t} + \nabla \cdot \mathbf{F} = \nabla \mu \nabla \mathbf{w}, \quad (20.49)$$

where

$$\mathbf{u} = \begin{Bmatrix} p/c^2 \\ v_i \end{Bmatrix}, \quad \mathbf{F}^j = \begin{Bmatrix} v_j \\ v_i v_j + p\delta_{ij} \end{Bmatrix}, \quad \mathbf{v} = \begin{Bmatrix} 0 \\ v_i \end{Bmatrix}. \quad (20.50)$$

Here p, c and v_i denote the pressure, (infinite) speed of sound and fluid velocity in direction x_i, respectively. The Jacobian matrices are given by

$$\mathbf{A}^x = \begin{Bmatrix} 0; & c^2; & 0; & 0; \\ 1; & 2u; & 0; & 0; \\ 0; & v; & u; & 0; \\ 0; & w; & 0; & u; \end{Bmatrix}, \quad \mathbf{A}^y = \begin{Bmatrix} 0 & 0; & c^2; & 0; \\ 0; & v; & u; & 0; \\ 1; & 0; & 2v; & 0; \\ 0; & 0; & w; & v; \end{Bmatrix},$$

$$\mathbf{A}^z = \begin{Bmatrix} 0; & 0; & 0; & 0; c^2 \\ 0; & w; & 0; & u; \\ 0; & 0; & w; & v; \\ 1; & 0; & 0; & 2w; \end{Bmatrix}, \quad (20.51)$$

and the $\mathbf{B}$ matrix is simply

$$\mathbf{B} = \begin{Bmatrix} 0; & 0; & 0; & 0; \\ 0; & 1; & 0; & 0; \\ 0; & 0; & 1; & 0; \\ 0; & 0; & 0; & 1; \end{Bmatrix}. \quad (20.52)$$

20.3.4. ADJOINT SOLVERS

The adjoint equations are of the form

$$\boldsymbol{\Psi}_{,t} - [\mathbf{A}^i]^T \boldsymbol{\Psi}_{,i} = \mathbf{B}^T \nabla\mu\nabla\boldsymbol{\Psi} - I^{\Omega}_{,\mathbf{u}}, \tag{20.53}$$

where, for the compressible Euler/Navier–Stokes case,

$$\boldsymbol{\Psi} = \begin{Bmatrix} \Psi_\rho \\ \Psi_{\rho v^i} \\ \Psi_{\rho e} \end{Bmatrix}, \quad \mathbf{A}^i = \mathbf{F}^i_{,\mathbf{u}}, \quad \mathbf{B} = \mathbf{w}_{,\mathbf{u}}. \tag{20.54}$$

This is a *linear* system of equations. In principle, any convergent solver can be employed to solve (20.53). The simplest of these are the explicit edge-based finite element solvers described in Chapter 10. At the beginning of the run, the steady flow results from the flow solver are read in and the Jacobian matrices are computed and stored. Consistent numerical fluxes are obtained using a blended second- and fourth-order edge-based dissipation. Naturally, other consistent numerical fluxes or implicit timestepping schemes can be employed, but the advantages of using more sophisticated schemes for the adjoint are not as clear as for the flow solvers.

20.3.4.1. Boundary conditions for the adjoint

As could be seen from (20.40) and (20.44) the derivative of the cost function with respect to the unknowns yields boundary conditions or source terms. The fact that one has to develop a new boundary condition for every new cost function is one of the more cumbersome aspects of this type of gradient evaluation. However, the benefits accrued are such that any (modest) investment in developing new boundary conditions for the adjoint quickly pays off. In the following, we list the boundary conditions required for the most common objective functions.

(a) *Inviscid forces*. The inviscid forces are given by

$$\mathbf{f} = \int_\Gamma p\mathbf{n}\, d\Gamma. \tag{20.55}$$

For a cost function of the form

$$I = \mathbf{f} \cdot \mathbf{c}_w, \tag{20.56}$$

where $\mathbf{c}_w = c^x_w, c^y_w, c^z_w$ represent weights, we obtain, from (20.39) and (20.44),

$$\mathbf{n} \cdot \boldsymbol{\Psi}_{\mathbf{v}} = -\mathbf{c}_w \cdot \mathbf{n}, \tag{20.57}$$

i.e. the *normal adjoint velocity is prescribed* while the tangential adjoint velocity is free to change. This condition is similar to the no-penetration boundary condition of inviscid flows. No condition is required for the adjoint pressure.

(b) *Prescribed pressure*. This condition is given by

$$I_{p_0} = \int_\Gamma (p - p_0)^2\, d\Gamma, \tag{20.58}$$

where p_0 denotes the prescribed pressure. From (20.39) and (20.44) this implies

$$\mathbf{n} \cdot \mathbf{\Psi}_{\mathbf{v}} = -2(p - p_0). \tag{20.59}$$

As before, the *normal adjoint velocity* is prescribed while the tangential adjoint velocity is free to change. No condition is required for the adjoint pressure.

(c) *Prescribed velocity*. This condition is given by

$$I_{v_0} = \int_\Gamma (\mathbf{v} - \mathbf{v}_0)^2 \, d\Gamma, \tag{20.60}$$

where $\mathbf{v}_0$ denotes the prescribed velocity. From (20.39) and (20.44) this implies

$$\mathbf{\Psi}_p = -2(\mathbf{v} - \mathbf{v}_0) \cdot \mathbf{n}, \tag{20.61}$$

i.e., the *adjoint pressure* is prescribed on the surfaces where the velocity is prescribed. No condition is required for the adjoint velocity.

20.3.5. GRADIENT EVALUATION

The gradient evaluation with respect to the design variables $\boldsymbol{\beta}$ may be separated into two parts using the chain rule:

$$\frac{\partial I}{\partial \beta_l} = \frac{\partial x_k^i}{\partial \beta_l} \frac{\partial I}{\partial x_k^i}. \tag{20.62}$$

The second part of this expression can be computed independently of the design variables chosen, and represents the gradient of the cost function with respect to the movement of the mesh points. This gradient (of length `ndimn*npoin`) is computed and stored. Let us consider this second expression in more detail. Given the solution of the Euler equations and the solution of the adjoint equations, the task is to obtain the gradients with respect to the movement of a point:

$$\frac{\partial I_L}{\partial x_k^i} = \left[\frac{\partial I}{\partial x_k^i} + \mathbf{\Psi}^T \frac{\partial \mathbf{R}}{\partial x_k^i} \right]. \tag{20.63}$$

This expression is easily evaluated using *finite differences*. Each point is moved in the x, y, z direction, the residual $\mathbf{R}$ is evaluated and the gradient is computed. In theory, a new residual evaluation would be required for each point, raising the cost to $2 \cdot 3 \cdot N_p$ evaluations (2 for central differences, 3 for the dimensions, N_p for the number of points). However, one can use the fact that even with higher-order schemes the points are connected by no more than nearest-neighbours (+1) to reduce the required number of residual evaluations. The points may be grouped (or coloured) in such a way that the points in each group are at least two neighbours away. This results in typically O(25–30) point groups, i.e. a total of O(150–180) residual evaluations. An outer loop over these groups or colours is performed. The next inner loop is over the dimensions (the movement directions). The next inner loop is over the positive and negative increments. The geometrical parameters are recalculated, and a timestep is performed using one step of an explicit solver. The increments in the unknowns are related to the residual by

$$\mathbf{M}_l \frac{\Delta \mathbf{u}}{\Delta t} = -\mathbf{R}, \tag{20.64}$$

where $\mathbf{M}_l$ denotes the lumped mass matrix. From this expression, the residuals are evaluated. Note that, using this procedure, the gradient can be computed in a 'black-box fashion', allowing the use of different flow solvers. For the evaluation of the finite differences, the movement of each point typically corresponds to 1% of the local element length.

20.4. Geometric constraints

The optimization process typically involves a compromise of many factors. The classic example cited so often for fluid dynamics is the reduction of drag. It is well known that, for a non-lifting body, the flat plate represents the optimum optimorum. Yet flat plates cannot carry any volume inside, a prime requirement for a device. Therefore, a compromise between volume and drag is required. Another example from aerodynamics is the wing. In the transonic regime, thickness again increases (wave) drag. A thin wing, on the other hand, requires a stronger structure, i.e. more weight. Fuel capacity may at some stage also become a problem. As before, a compromise between volume and performance must be reached. The volume constraint is one of many possible *geometric constraints*. Other constraints of this kind include:

- prescribed (minimum/maximum) volume;
- prescribed (minimum/maximum) cross-sectional area;
- prescribed (minimum/maximum) sectional thickness;
- prescribed (minimum/maximum) deviation from original shape;
- prescribed (minimum/maximum) surface curvature.

These constraints may be treated in a variety of ways:

- by inclusion in the cost function;
- by projection of the gradient;
- by post-processing the new shape; etc.

We treat each one of these for the volume constraint,

$$V(\boldsymbol{\beta}) = V_0. \tag{20.65}$$

20.4.1. VOLUME CONSTRAINT VIA COST FUNCTION

The volume constraint given by (20.65) may be added to the cost function in the form

$$I = I^0 + I^V = I^0 + c_V\left[\left(\frac{V - V_O}{V_O}\right)^2\right]^q, \tag{20.66}$$

where c_V and V_O denote a weight factor and a reference volume, respectively, and I_0 is the unconstrained cost function. The derivative is given by

$$I^V_{,x} = c_V 2q\left[\left(\frac{V - V_O}{V_O}\right)^2\right]^{q-\frac{1}{2}} \frac{\operatorname{sgn}(V - V_O)}{V_O} V_{,x}. \tag{20.67}$$

Observe that, for the common choice $q = 1/2$, the first term becomes unity, i.e. there is no effect from the deviation from the desired volume V_0, but the gradient simply flips from positive to negative with a constant weight factor. Choosing $q = 1$ or $q = 0.75$ yields a smooth transition. Numerically, the volume can be evaluated by:

- summing up the volumes of all elements (and subtracting it from a reference volume);
- utilizing the divergence theorem.

We discuss the second option in more detail. The divergence theorem states that

$$\int_\Omega \nabla \cdot \mathbf{v} \, d\Omega = \int_\Gamma \mathbf{v} \cdot \mathbf{n} \, d\Gamma. \tag{20.68}$$

Given that the volume is $\int_\Omega d\Omega$, we desire $\nabla \cdot \mathbf{v} = 1$, which can be obtained in a variety of ways, e.g. $\mathbf{v} = (x, y, z)/3$. We therefore have

$$V = \frac{1}{3} \int_\Gamma (x, y, z) \cdot \mathbf{n} \, d\Gamma. \tag{20.69}$$

This expression can be separated into x, y, z components, yielding

$$V = \frac{1}{3} \sum_{id=1,3} V_{id} = \frac{1}{3} \sum_{id=1,3} \int_\Gamma x_{id} n_{id} \, d\Gamma. \tag{20.70}$$

The separation into components is useful in cases with planes of symmetry or unclosed objects, where one or more of the V_{id} components are incomplete. Consider the evaluation of the gradient of the volume cost factor with respect to the movement of an arbitrary (surface) gridpoint $\mathbf{x}_i$. In finite difference form, this yields

$$\frac{\Delta V}{\Delta \mathbf{x}_i^k} = \frac{1}{\Delta \mathbf{x}_i^k} \sum_{jf \text{ in } i} \left[\int_\Gamma x_{id} n_{id} \, d\Gamma \bigg|_{\mathbf{x}_i^k + \Delta \mathbf{x}_i^k} - \int_\Gamma x_{id} n_{id} \, d\Gamma \bigg|_{\mathbf{x}_i^k - \Delta \mathbf{x}_i^k} \right]. \tag{20.71}$$

20.4.2. VOLUME CONSTRAINT VIA GRADIENT PROJECTION

The volume constraint given by (20.65) may be rewritten as

$$\delta V = \delta \boldsymbol{\beta}^T \cdot V_{,\boldsymbol{\beta}} = 0, \tag{20.72}$$

implying that the change of $\boldsymbol{\beta}$ in the direction $V_{,\boldsymbol{\beta}}$ must vanish. For n constraints of the form $V^i(\boldsymbol{\beta}) = V_0^i(\boldsymbol{\beta})$ we define the n gradients: $\mathbf{g}_i^c = V_{,\boldsymbol{\beta}}^i$, $i = 1, n$. The imposition of constraints is then accomplished in two steps.

Step 1. Orthogonalization of gradients:

```
do:  for $i = 1, n$
   $\mathbf{g}_i^{c'} = \mathbf{g}_i^c \,/\, |\mathbf{g}_i^c|$
   do:  for $j = i + 1, n$
      $\mathbf{g}_j^c = \mathbf{g}_j^c - (\mathbf{g}_j^c \cdot \mathbf{g}_i^{c'})\, \mathbf{g}_i^{c'}$
   enddo
enddo
```

Step 2. Projection of cost function gradient:

$$I_{,\boldsymbol{\beta}} = I_{,\boldsymbol{\beta}} - (I_{,\boldsymbol{\beta}} \cdot \mathbf{g}_i^{c'}) \mathbf{g}_i^{c'}.$$

20.4.3. VOLUME CONSTRAINT VIA POST-PROCESSING

In this case, the volumes before and after design changes are compared. The surface is then moved in the normal direction in such a way as to impose (20.65).

20.5. Approximate gradients

As seen before, the complete gradient is composed of two parts:

- gradient with respect to the geometry;
- gradient with respect to the state.

So

$$\frac{\partial I_L}{\partial x_k^i} = \left[\frac{\partial I}{\partial x_k^i} + \mathbf{\Psi}^T \frac{\partial \mathbf{R}}{\partial x_k^i} \right]. \tag{20.73}$$

It has been repeatedly demonstrated that in many cases the gradient with respect to the geometry is much larger than the gradient with respect to the state (Mohammadi (1997), Medic *et al.* (1998), Mohammadi (1999), Soto and Löhner (2000), Mohammadi and Pironneau (2001)). This obviates the need for an adjoint solver, as

$$\frac{\partial I_L}{\partial x_k^i} \approx \frac{\partial I}{\partial x_k^i}, \tag{20.74}$$

and is referred to as *approximate gradient evaluation*.

20.6. Multipoint optimization

Shapes that are optimal for one given flow (angle of attack, speed, height) seldomly work well across a wide range of flight conditions (Drela (1998), Huyse and Lewis (2001), Li *et al.* (2001)). The classical example of this phenomenon is the Korn airfoil, which exhibits a shock-free behaviour at a certain Mach number and angle of attack. However, just changing slightly the Mach number or the angle of attack leads to the appearance of very strong shocks, making this type of airfoil unsuitable for airplanes. Is has been observed (Jameson (1988), Reuther (1999), Huyse and Lewis (2001), Li *et al.* (2001)) that the best way to steer a design away from such singular behaviour is to conduct a so-called *multipoint optimization*. In this case, the design has to produce a favourable cost function for several flight conditions. This can be cast in the form

$$I = \sum_i \gamma_i I(Ma_i, \alpha_i), \tag{20.75}$$

where I denotes the original cost function and Ma_i, α_i the flight conditions for which the multipoint design is carried out. Several schemes have been proposed to choose the weights γ_i (Li *et al.* (2001)), although in most cases they have been kept constant during optimization. Note that this is an additive cost function of different flight conditions (states), implying that gradient evaluations are also additive. The flow and adjoint solutions have to be computed for each flight condition. Given that the CPU requirement of each one of these solutions is similar, this lends itself naturally to parallel processing. A current topic of research is concerned with the optimal (smallest) number of design points required to ensure a so-called *robust design* (Li *et al.* (2001)).

20.7. Representation of surface changes

The representation of surface changes has a direct effect on the number of design iterations required, as well as the shape that may be obtained through optimal shape design. In general, the number of design iterations increases with the number of design variables, as does the possibility of 'noisy designs' due to a richer surface representation space.

One can broadly differentiate two classes of surface change representation: direct and indirect. In the first case, the design changes are directly computed at the nodes representing the surface (line points, Bezier points, Hicks–Henne (Hicks and Henne (1978)) functions, a set of known airfoils/wings/hulls, discrete surface points, etc.). In the second case, a (coarse) set of patches is superimposed on the surface (or embedded in space). The surface change is then defined on this set of patches, and subsequently translated to the CAD representation of the surface (see Figure 20.9).

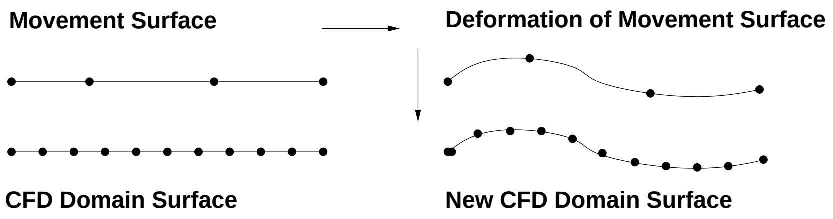


Figure 20.9. Indirect surface change representation

20.8. Hierarchical design procedures

Optimal shape design (and optimization in general) may be viewed as an information building process. During the optimization process, more and more information is gained about the design space and the consequences design changes have on the objective function. Consider the case of a wing for a commercial airplane. At the start, only global objectives, measures and constraints are meaningful: take-off weight, fuel capacity, overall measures, sweep angle, etc. (Raymer (1999)). At this stage, it would be foolish to use a RANS or LES solver to determine the lift and drag. A neural net or a lifting line theory yield sufficient flow-related information for these global objectives, measures and constraints. As the design matures, the information shifts to more local measures: thickness, chamber, twist angle, local wing stiffness, etc. The flowfield prediction needs to be upgraded to either lifting line theory, potential flow or Euler solvers. During the final stages, the information reaches the highest precision. It is here that RANS or LES solvers need to be employed for the flow analysis/prediction. This simple wing design case illustrates the basic principle of hierarchical design. Summarizing, the key idea of hierarchical design procedures is to *match the available information* of the design space to:

- the number of design variables;
- the sophistication of the physical description; and
- the discretization used.

Hierarchical in this context means that the addition of further degrees of freedom does not affect in a major way the preceding ones. A typical case of a hierarchical representation is the Fourier series for the approximation of a function. The addition of further terms in the series does not affect the previous ones. Due to the nonlinearity of the physics, such perfect orthogonality is difficult to achieve for the components of optimal shape design (design variables, physical description, discretization used).

In order to carry out a hierarchical design, all the components required for optimal shape design: design variables, physical description and discretization used must be organized hierarchically.

The *number of design variables* has to increase from a few to possibly many ($>10^3$), and in such a way that the addition of more degrees of freedom do not affect the previous ones. A very impressive demonstration of this concept was shown by Marco and Beux (1993) and Kuruvila *et al.* (1995). The hierarchical storage of analytical or discrete surface data has been studied in Popovic and Hoppe (1997).

The *physical representation* is perhaps the easiest to organize. For the flowfield, complexity and fidelity increase in the following order: lifting line, potential, potential with boundary layer, Euler, Euler with boundary layer, RANS, LES/DNS, etc. (Alexandrov *et al.* (2000), Peri and Campana (2003), Yang and Löhner (2004)). For the structure, complexity and fidelity increase in the following order: beam theory, shells and beams, solids.

The *discretization used* is changed from coarse to fine for each one of the physical representations chosen as the design matures. During the initial design iterations, coarser grids and/or simplified gradient evaluations (Dadone and Grossman (2003), Peri and Campana (2003)) can be employed to obtain trends. As the design matures for a given physical representation, the grids are progressively refined (Kuruvila *et al.* (1995), Dadone and Grossman (2000), Dadone *et al.* (2000), Dadone (2003)).

20.9. Topological optimization via porosities

A formal way of translating the considerable theoretical and empirical legacy of topological optimization found in structural mechanics (Bendsoe and Kikuchi (1988), Jakiela *et al.* (2000), Kicinger *et al.* (2005)) is via the concept of porosities. Let us recall the basic topological design procedure employed in structural dynamics: starting from a ‘design space’ or ‘design volume’ and a set of applied loads, remove the parts (regions, volumes) that do not carry any significant loads (i.e. are stress-free), until the minimum weight under stress constraints is reached. It so happens that one of the most common design objectives in structural mechanics is the minimization of weight, making topological design an attractive procedure for preliminary design. Similar ideas have been put forward for fluid dynamics (Borrvall and Peterson (2003), Moos *et al.* (2004), Hassine *et al.* (2004), Guest and Prévost (2006), Othmer *et al.* (2006)) as well as heat transfer (Hassine *et al.* (2004)). The idea is to remove, from the flowfield, regions where the velocity is very low, or where the residence time of particles is considerable (recirculation zones). The mathematical foundation of all of these methods is the so-called topological derivative, which relates the change in the cost function(s) to a small change in volume. For classic optimal shape design, applying this derivative at the boundary yields the desired gradient with respect to the design variables. The removal of available volume or ‘design space’ from the flowfield can be re-interpreted as an increase in the porosity π in the flowfield:

$$\rho \mathbf{v}_{,t} + \rho \mathbf{v}\nabla\mathbf{v} + \nabla p = \nabla\mu\nabla\mathbf{v} - \pi\mathbf{v}. \tag{20.76}$$

Note that no flow will occur in regions of large porosities π. If we consider the porosity as the design variable, we have, from (20.35), after solution for the adjoint $\mathbf{\Psi}$,

$$\delta I = [I_{,\pi} + \mathbf{\Psi}^T \mathbf{R}_{,\pi}]\delta\pi. \tag{20.77}$$

Observe, however, that analytically and to first order,

$$\mathbf{R}_{,\pi} = \mathbf{v}. \tag{20.78}$$

If we furthermore assume that the objective function involves boundary integrals (lift, drag, total loss between in- and outlets, etc.) then $I_{,\pi} = 0$ and we have

$$\delta I = \mathbf{\Psi}^T \cdot \mathbf{v}\delta\pi. \tag{20.79}$$

This implies that if we desire to remove the volume regions that will have the least (negative) effect on the objective function, we should proceed from those regions where either the adjoint velocity vanishes or where the velocity vanishes. If the adjoint is unavailable, then we can only proceed with the second alternative. This simplified explanation at least makes plausible the fact that the strategy of removing from the flowfield the regions of vanishing velocity by increasing the porosity there can work. Indeed, a much more detailed mathematical analysis (Hassine *et al.* (2004)) reveals that this is indeed the case for the Stokes limit ($Re \to 0$).

20.10. Examples

20.10.1. DAMAGE ASSESSMENT FOR CONTAMINANT RELEASE

The intentional or unintentional release of hazardous materials can lead to devastating consequences. Assuming that the amount of contaminant is finite and that the population density in the region of interest is given, for any given meteorological condition the location of the release becomes the main input variable. Damage as a function of release location can have many local extrema, as pockets of high concentration can linger in recirculation zones or diffuse slowly while being transported along street canyons. For this reason, genetic algorithms offer a viable optimization tool for this process design. The present example, taken from Camelli and Löhner (2004), considers the release in an area representative of an inner city composed of three by two blocks. The geometry definition and the surface mesh are shown in Figure 20.10(a). Each of the fitness/damage evaluations (i.e. dispersion simulation runs) took approximately 80 minutes using a PC with Intel P4 chip running at 2.53 GHz with 1 Gbyte RAM, Linux OS and Intel compiler.

Three areas of release were studied (see Figure 20.10(b)): the upwind zone of the complex of buildings, the street level and the core of one of the blocks. In all cases, the height of release was set to $z = 1.5$ m. The genetic optimization was carried out for 20 generations, with two chromosomes (x/y location of release point) and 10 individuals in the population.

The location and associated damage function for each of the releases computed during the optimization process are shown in Figures 20.11(a)–(d). Interestingly, the maximum

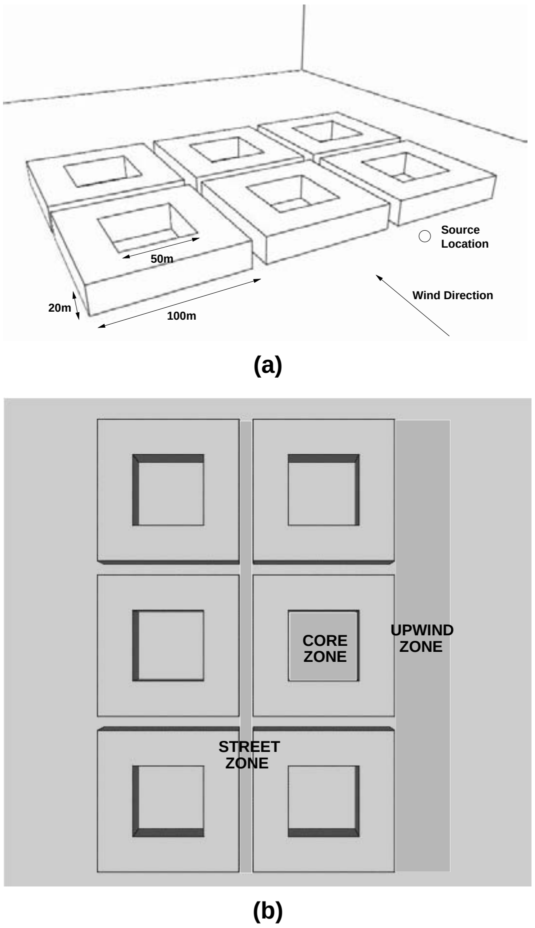


Figure 20.10. (a) Problem definition; (b) zones considered for release;

damage is produced in the street area close to one of the side corners of the blocks. The cloud (Figures 20.11(e)–(i)) shows the suction effect produced by the streets that are in the direction of the wind, allowing for a very long residence time of contaminants close to the ground for this particular release location.

20.10.2. EXTERNAL NOZZLE

This example, taken from Soto *et al.* (2004), considers an external nozzle, typical of those envisioned for the X-34 airplane. There are no constraints on the shape. The objective is to maximize the thrust of the nozzle. The flow conditions are as follows:

- inflow (external flowfield): $Ma = 3.00$, $\rho = 0.5$, $v = 1.944$, $pr = 0.15$, $\alpha = 0.0°$;
- nozzle exit: $Ma = 1.01$, $\rho = 1.0$, $v = 1.000$, $pr = 0.18$, $\alpha = -45.0°$.

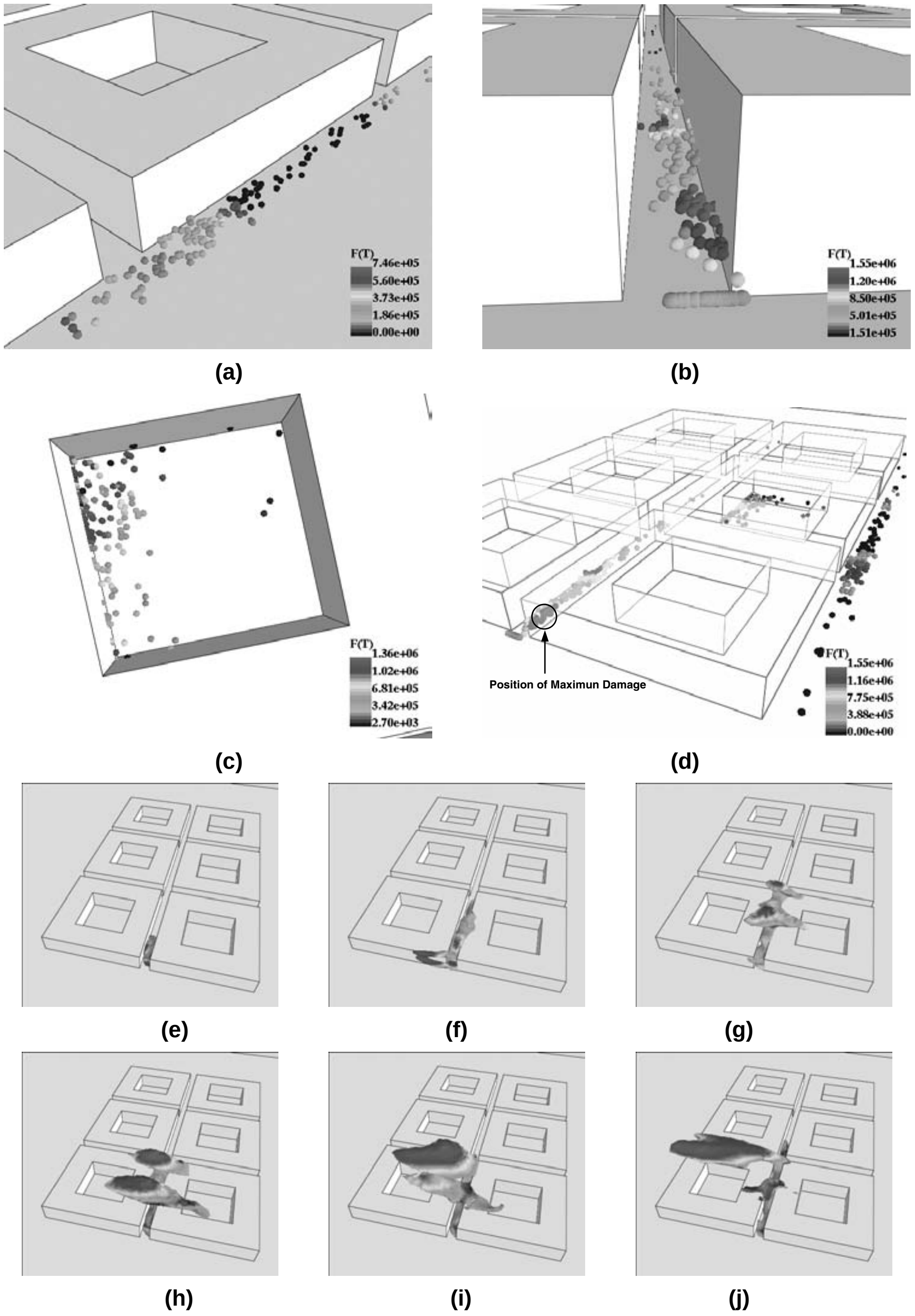


Figure 20.11. (a)–(d) Release positions coloured according to damage; (e)–(i) evolution of maximum damage cloud

Although this is in principle a 2-D problem, the case was run in three dimensions. The mesh had approximately 51 000 points and 267 000 elements. The total number of design variables was 918. Thus, the only viable choice was given by the adjoint methodology outlined above. Figures 20.12(a)–(f) show the initial and final mach numbers and pressures, as well as the evolution of the shape and the thrust.

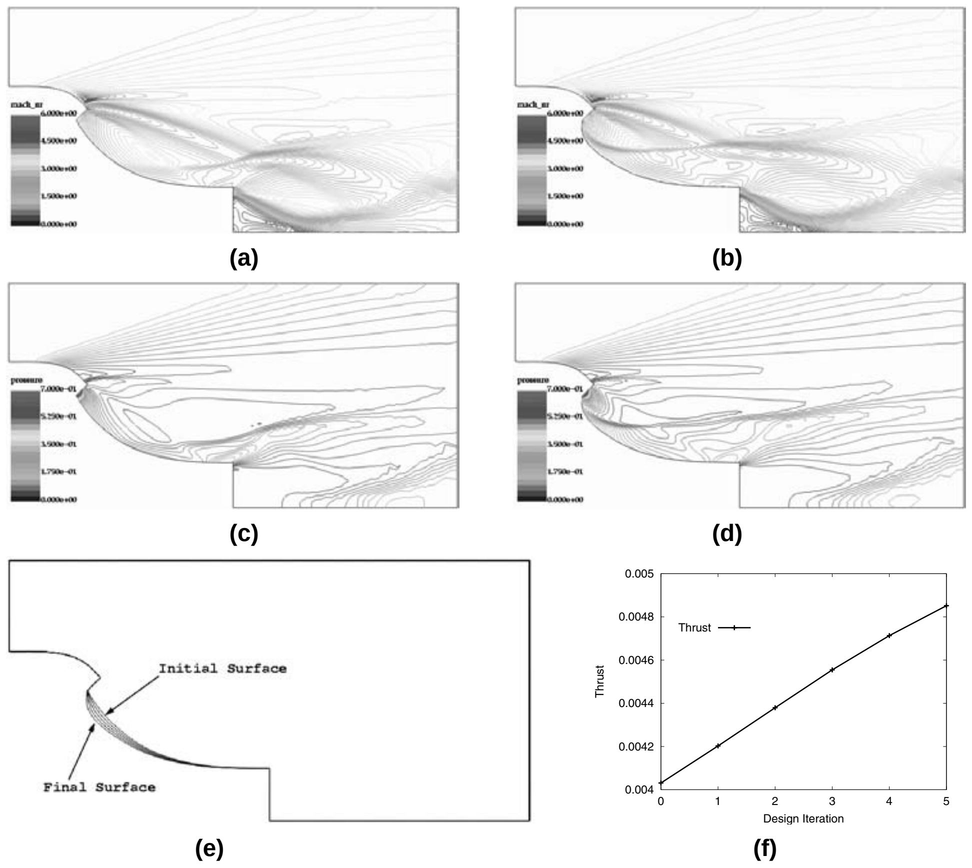
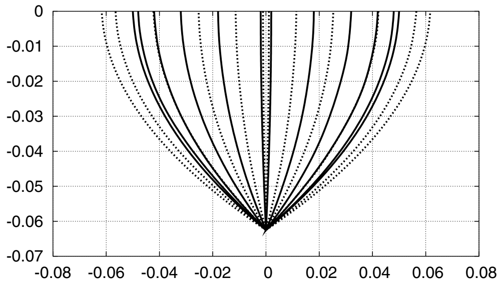


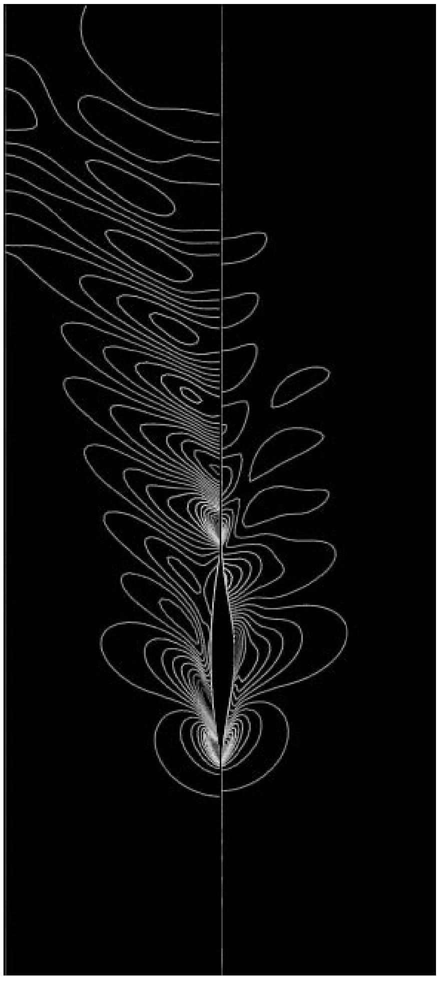
Figure 20.12. External nozzle: Mach numbers for (a) first and (b) last shape; pressures for (c) first and (d) last shape; evolution of (e) shape and (f) thrust

20.10.3. WIGLEY HULL

This example, taken from Yang and Löhner (2004), shows the use of an indirect surface representation as well as the finite difference evaluation of gradients via reduced complexity models. The geometry considered is a Wigley hull. The hull length and displacement are fixed while the wave drag is minimized during the optimization process. The water line is represented by a six-point B-spline. Four of the six spline points are allowed to change and they are chosen as design variables. The hull surface, given by a triangulation, is then modified according to these (few) variables. The gradients of the objective function with

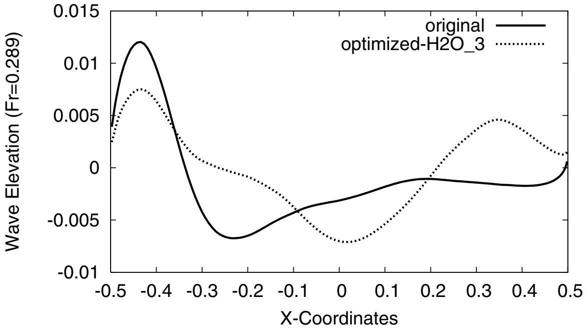


(a)



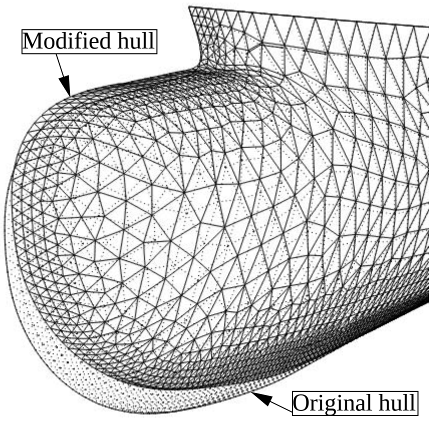
(b)

Figure 20.13. Wigley hull: (a) comparison of frame lines (solid, original; dashed, optimized); (b) comparison of wave patterns generated (left, original; right, optimized); (c) comparison of wave profiles (solid, original; dashed, optimized)

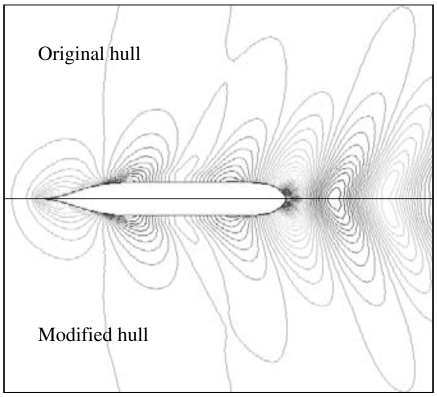


(c)

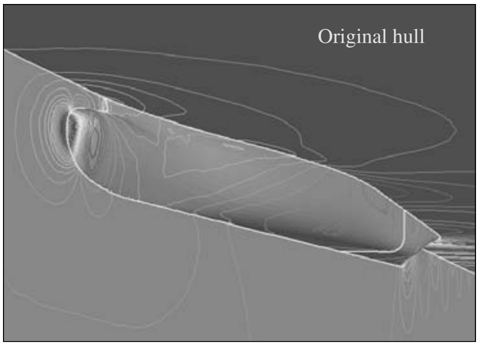
Figure 20.13. Continued



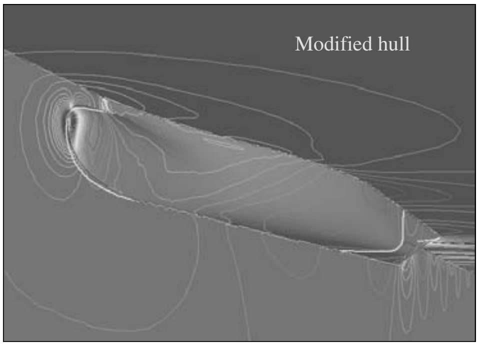
(a)



(b)



(c)



(d)

Figure 20.14. KCS: (a) surface mesh; (b) wave pattern; pressure contours of (c) the original hull; and (d) the modified hull

respect to the design variables are obtained via finite differences using a fast potential flow solver. Whenever the design variables are updated, the cost function is re-evaluated using an Euler solver with free surface fitting. Given that the potential flow solver is two orders of magnitude faster than the Euler solver, the evaluation of gradients via finite differences does not add a significant computational cost to the overall design process. Figures 20.13(a)–(c) compare the frame lines and wave patterns generated for the original and optimized hulls at a Froude number of $Fr = 0.289$. While the wetted surface and displacement remain almost unchanged, more than 50% wave drag reduction is achieved with the optimized hull. The optimized hull has a reduced displacement in both the bow and stern regions and an increased displacement in the middle of the hull.

20.10.4. KRISO CONTAINER SHIP (KCS)

This example, taken from Löhner *et al.* (2003), considers a modern container ship with bulb bow and stern. The objective is to modify the shape of the bulb bow in order to reduce the wave drag. The Froude number was set to $Fr = 0.25$, and no constraints were imposed on the shape. The volume mesh had approximately 100 000 points and 500 000 tetrahedra, and the free surface had approximately 10 000 points and 20 000 triangles. The number of design variables was in excess of 200, making the adjoint approach the only one viable. Figure 20.14(a) shows the surface mesh in the bulb region for the original and final designs obtained. Figure 20.14(b) shows the comparison of wave patterns generated by the original and final hull forms. The wave drag reduction was of the order of 4%. Figures 20.14(c) and (d) show the pressure contours on the original and modified hulls.

REFERENCES

Aftosmis, M. A Second-Order TVD Method for the Solution of the 3-D Euler and Navier–Stokes Equations on Adaptively Refined Meshes; *Proc. 13th Int. Conf. Num. Meth. Fluid Dyn.*, Rome, Italy (1992).

Aftosmis, M. and N. Kroll. A Quadrilateral Based Second-Order TVD Method for Unstructured Adaptive Meshes; *AIAA*-91-0124 (1991).

Aftosmis, M. and J. Melton. Robust and Efficient Cartesian Mesh Generation from Component-Based Geometry; *AIAA*-97-0196 (1997).

Aftosmis, M., M.J. Berger and G. Adomavicius. A Parallel Multilevel Method for Adaptively Refined Cartesian Grids with Embedded Boundaries; *AIAA*-00-0808 (2000).

Aftosmis, M., M.J. Berger and J. Alonso. Applications of a Cartesian Mesh Boundary-Layer Approach for Complex Configurations; *AIAA*-06-0652 (2006).

Ainsworth, M., J.Z. Zhu, A.W. Craig and O.C. Zienkiewicz. Analysis of the Zienkiewicz-Zhu a Posteriori Error Estimate in the Finite Element Method; *Int. J. Num. Meth. Eng.* 28(12), 2161–2174 (1989).

Alessandrini, B. and G. Delhommeau. A Multigrid Velocity-Pressure- Free Surface Elevation Fully Coupled Solver for Calculation of Turbulent Incompressible Flow Around a Hull; *Proc. 21st Symp. on Naval Hydrodynamics*, Trondheim, Norway, June (1996).

Alexandrov, N., R. Lewis, C. Gumbert, L. Green and P. Newman. Optimization with Variable Resolution Models Applied to Aerodynamic Wing Design; *AIAA*-00-0841 (2000).

Allwright, S. Multiblock Topology Specification and Grid Generation for Complete Aircraft Configurations; *AGARD-CP-464*, 11 (1990).

Alonso, J., L. Martinelli and A. Jameson. Multigrid Unsteady Navier–Stokes Calculations with Aeroelastic Applications; *AIAA*-95-0048 (1995).

Anderson, J.D.A. *Computational Fluid Dynamics: The Basics with Applications;* McGraw-Hill (1995).

Anderson, W.K. and D.L. Bonhaus. Navier–Stokes Computations and Experimental Comparisons for Multielement Airfoil Configurations; *AIAA*-93-0645 (1993).

Anderson, W.K. and V. Venkatakrishnan. Aerodynamic Design Optimization on Unstructured Meshes With a Continuous Adjoint Formulation; *AIAA*-97-0643 (1997).

Anderson, D.A., J.C. Tannehill and R.H. Pletcher. *Computational Fluid Mechanics and Heat Transfer;* McGraw-Hill (1984).

Angot, P., C.-H. Bruneau and P. Fabrie. A Penalization Method to Take Into Account Obstacles in Incompressible Viscous Flows; *Numerische Mathematik* 81, 497–520 (1999).

Arcilla, A.S., J. Häuser, P.R. Eiseman and J.F. Thompson (eds.). *Proc. 3rd Int. Conf. Numerical Grid Generation in Computational Fluid Dynamics and Related Fields;* North-Holland (1991).

Atkins, H. and C.-W. Shu. Quadrature-Free Implementation of the Discontinuous Galerkin Method for Hyperbolic Equations; *AIAA*-96-1683 (1996).

Baaijens, F.P.T. A Fictitious Domain/Mortar Element Method for Fluid-Structure Interaction; *Int. J. Num. Meth. Fluids* 35, 734–761 (2001).

Babuska, I., J. Chandra and J.E. Flaherty (eds.). *Adaptive Computational Methods for Partial Differential Equations;* SIAM, Philadelphia (1983).

Babuska, I., O.C. Zienkiewicz, J. Gago and E.R. de A. Oliveira (eds.). *Accuracy Estimates and Adaptive Refinements in Finite Element Computations;* John Wiley & Sons (1986).

Baehmann, P.L., M.S. Shepard and J.E. Flaherty. A Posteriori Error Estimation for Triangular and Tetrahedral Quadratic Elements Using Interior Residuals; *Int. J. Num. Meth. Eng.* 34, 979–996 (1992).

Baker, T.J. Three-Dimensional Mesh Generation by Triangulation of Arbitrary Point Sets; *AIAA-CP*-87-1124, *8th CFD Conf.*, Hawaii (1987).

Baker, T.J. Developments and Trends in Three-Dimensional Mesh Generation. *Appl. Num. Math.* 5, 275–304 (1989).

Balaras, E. Modeling Complex Boundaries Using an External Force Field on Fixed Cartesian Grids in Large-Eddy Simulations; *Comp. Fluids* 33, 375–404 (2004).

Baldwin, B.S. and H. Lomax. Thin Layer Approximation and Algebraic Model for Separated Turbulent Flows; *AIAA*-78–257 (1978).

Barth, T. A 3-D Upwind Euler Solver for Unstructured Meshes *AIAA*-91-1548-CP (1991).

Barth, T. Steiner Triangulation for Isotropic and Stretched Elements; *AIAA*-95-0213 (1995).

Batina, J.T. Vortex-Dominated Conical-Flow Computations Using Unstructured Adaptively-Refined Meshes; *AIAA J.* 28(11), 1925–1932 (1990a).

Batina, J.T. Unsteady Euler Airfoil Solutions Using Unstructured Dynamic Meshes; *AIAA J.* 28(8), 1381–1388 (1990b).

Batina, J. A Gridless Euler/Navier–Stokes Solution Algorithm for Complex Aircraft Configurations; *AIAA*-93-0333 (1993).

Baum, J.D. and R. Löhner. Numerical Simulation of Shock-Elevated Box Interaction Using an Adaptive Finite Element Shock Capturing Scheme; *AIAA*-89-0653 (1989).

Baum, J.D. and R. Löhner. Numerical Simulation of Shock Interaction with a Modern Main Battlefield Tank; *AIAA*-91-1666 (1991).

Baum, J.D. and R. Löhner. Numerical Simulation of Passive Shock Deflector Using an Adaptive Finite Element Scheme on Unstructured Grids; *AIAA*-92-0448 (1992).

Baum, J.D. and R. Löhner. Numerical Simulation of Pilot/Seat Ejection from an F-16; *AIAA*-93-0783 (1993).

Baum, J.D. and R. Löhner. Numerical Simulation of Shock-Box Interaction Using an Adaptive Finite Element Scheme; *AIAA J.* 32(4), 682–692 (1994).

Baum, J.D., H. Luo and R. Löhner. Numerical Simulation of a Blast Inside a Boeing 747; *AIAA*-93-3091 (1993).

Baum, J.D., H. Luo and R. Löhner. A New ALE Adaptive Unstructured Methodology for the Simulation of Moving Bodies; *AIAA*-94-0414 (1994).

Baum, J.D., H. Luo and R. Löhner. Numerical Simulation of Blast in the World Trade Center; *AIAA*-95-0085 (1995a).

Baum, J.D., H. Luo and R. Löhner. Validation of a New ALE, Adaptive Unstructured Moving Body Methodology for Multi-Store Ejection Simulations; *AIAA*-95-1792 (1995b).

Baum, J.D., H. Luo, R. Löhner, C. Yang, D. Pelessone and C. Charman. A Coupled Fluid/Structure Modeling of Shock Interaction with a Truck; *AIAA*-96-0795 (1996).

Baum, J.D., R. Löhner, T.J. Marquette and H. Luo. Numerical Simulation of Aircraft Canopy Trajectory; *AIAA*-97-1885 (1997a).

Baum, J.D., H. Luo, R. Löhner, E. Goldberg and A. Feldhun. Application of Unstructured Adaptive Moving Body Methodology to the Simulation of Fuel Tank Separation From an F-16 C/D Fighter; *AIAA*-97-0166 (1997b).

Baum, J.D., H. Luo and R. Löhner. The Numerical Simulation of Strongly Unsteady Flows With Hundreds of Moving Bodies; *AIAA*-98-0788 (1998).

Baum, J.D., H. Luo, E. Mestreau, R. Löhner, D. Pelessone and C. Charman. A Coupled CFD/CSD Methodology for Modeling Weapon Detonation and Fragmentation; *AIAA*-99-0794 (1999).

Baum, J.D., E. Mestreau, H. Luo, D. Sharov, J. Fragola and R. Löhner. CFD Applications in Support of the Space Shuttle Risk Assessment; *JANNAF* (2000).

Baum, J.D., E. Mestreau, H. Luo, R. Löhner, D. Pelessone and Ch. Charman. Modeling Structural Response to Blast Loading Using a Coupled CFD/CSD Methodology; *Proc. Des. An. Prot. Struct. Impact/Impulsive/Shock Loads (DAPSIL)*, Tokyo, Japan, December (2003).

Baum, J.D., E. Mestreau, H. Luo, R. Löhner, D. Pelessone, M.E. Giltrud and J.K. Gran. Modeling of Near-Field Blast Wave Evolution; *AIAA*-06-0191 (2006).

Beam, R.M. and R.F. Warming. An Implicit Finite Difference Algorithm for Hyperbolic Systems in Conservation-Law Form; *J. Comp. Phys.* 22, 87–110 (1978).

Becker, R. and R. Rannacher. An Optimal Control Approach to Error Control and Mesh Adaptation; in: A. Iserles (ed.), *Acta Numerica 2001*, Cambridge University Press (2001).

Bell, J.B. and D.L. Marcus. A Second-Order Projection Method for Variable-Density Flows; *J. Comp. Phys.* 101, 2 (1992).

Bell, J.B., P. Colella and H. Glaz. A Second-Order Projection Method for the Navier–Stokes Equations; *J. Comp. Phys.* 85, 257–283 (1989).

Belytchko, T. and T.J.R. Hughes (eds.). *Computer Methods for Transient Problems*, North-Holland, Dordrecht (1983).

Belytschko, T., Y. Lu and L. Gu. Element Free Galerkin Methods; *Int. J. Num. Meth. Eng.* 37, 229–256 (1994).

Bendsoe, M.P. and N. Kikuchi. Generating Optimal Topologies in Structural Design Using a Homogenization Method; *Comp. Meth. Appl. Mech. Eng.* 71, 197–224 (1988).

Bendsoe, M.P. and O. Sigmund. *Topology Optimization: Theory, Methods and Applications;* Springer-Verlag, Berlin (2004).

Benek, J.A., P.G. Buning and J.L. Steger. A 3-D Chimera Grid Embedding Technique; *AIAA*-85-1523 (1985).

Berger, M.J. and R.J. LeVeque. An Adaptive Cartesian Mesh Algorithm for the Euler Equations in Arbitrary Geometries; *AIAA*-89-1930 (1989).

Berger, M.J. and J. Oliger. Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations; *J. Comp. Phys.* 53, 484–512 (1984).

Bertrand, F., P.A. Tanguy and F. Thibault. Three-Dimensional Fictitious Domain Method for Incompressible Fluid Flow Problems; *Int. J. Num. Meth. Fluids* 25, 136–719 (1997).

Berz, M., C.H. Bischof, G.F. Corliss and A. Griewank (eds.). *Computational Differentiation: Applications, Techniques and Tools;* SIAM, Philadelphia (1996).

Besnard, E., A. Schmitz, K. Kaups, G. Tzong, H. Hefazi, O. Kural, H. Chen and T. Cebeci. Hydrofoil Design and Optimization for Fast Ships; *Proc. 1998 ASME Int. Cong. and Exhibition*, Anaheim, CA, Nov. (1998).

Biausser, B. P. Fraunie, S. Grilli and R. Marcer. Numerical Analysis of the Internal Kinematics and Dynamics of Three-Dimensional Breaking Waves on Slopes; *Int. J. Offshore and Polar Eng.* 14(4), 247–256 (2004).

Bieterman, M.B., J.E. Bussoletti, G.L. Hilmes, F.T. Johnson, R.G. Melvin and D.P. Young. An Adaptive Grid Method for Analysis of 3-D Aircraft Configurations; *Comp. Meth. Appl. Mech. Eng.* 101, 225–249 (1992).

Bijl, H. M.H. Carpenter, V.N Vatsa and C.A. Kennedy. Implicit Time Integration Schemes for the Unsteady Compressible Navier–Stokes Equations: Laminar Flow; *J. Comp. Phys.* 179(1), 313–329 (2002).

Billey, V., J. Périaux, P. Perrier, B. Stoufflet. 2-D and 3-D Euler Computations with Finite Element Methods in Aerodynamic; *International Conference on Hypersonic Problems*, Saint-Etienne, January 13–17 (1986).

Bischof, C., A. Carle, G. Corliss, A. Griewank and P. Hovland. ADIFOR - Generating Derivative Codes from Fortran Programs; *Scientific Programming* 1, 1–29 (1992).

Biswas, R. and R. Strawn. A New Procedure for Dynamic Adaption of Three-Dimensional Unstructured Grids; *AIAA*-93-0672 (1993).

Blacker, T.D. and M.B. Stephenson. Paving: A New Approach to Automated Quadrilateral Mesh Generation; *Int. J. Num. Meth. Eng.* 32, 811–847 (1992).

Blazek, J. *Computational Fluid Dynamics: Principles and Applications;* Elsevier Science (2001).

Boender, E. Reliable Delaunay-Based Mesh Generation and Mesh Improvement; *Comm. Appl. Num. Meth. Eng.* 10, 773–783 (1994).

Bonet, J. and J. Peraire. An Alternate Digital Tree Algorithm for Geometric Searching and Intersection Problems; *Int. J. Num. Meth. Eng.* 31, 1–17 (1991).

Book, D.L. (ed.). *Finite-Difference Techniques for Vectorized Fluid Dynamics Calculations;* Springer-Verlag (1981).

Book, D.L., J.P. Boris and K. Hain. Flux-corrected Transport. II. Generalizations of the Method; *J. Comp. Phys.* 18, 248–283 (1975).

Boris, J.P. and D.L. Book. Flux-corrected Transport. I. SHASTA, a Transport Algorithm that works; *J. Comp. Phys.* 11, 38–69 (1973).

Boris, J.P. and D.L. Book. Flux-corrected Transport. III. Minimal-Error FCT Algorithms; *J. Comp. Phys.* 20, 397–431 (1976).

Borrvall, T. and J. Peterson. Topology Optimization of Fluids in Stokes Flow; *Int. J. Num. Meth. Fluids* 41, 77–107 (2003).

Boschitsch, A.H. and T.R. Quackenbush. High Accuracy Computations of Fluid-Structure Interaction in Transonic Cascades; *AIAA*-93-0485 (1993).

Bowyer, A. Computing Dirichlet Tessellations; *The Computer Journal* 24(2), 162–167 (1981).

Brackbill, J.U. and J.S. Saltzman. Adaptive Zoning for Singular Problems in Two Dimensions; *J. Comp. Phys.* 46, 342–368 (1982).

Brand, K. *Multigrid Bibliography* (3rd Edn); GMD, Bonn, W. Germany (1983).

Bridson, R., J. Teran, N. Molino and R. Fedkiw. Adaptive Physics Based Tetrahedral Mesh Generation Using Level Sets; *Engineering with Computers* (2005).

Briley, W.R. and H. McDonald. Solution of the Multi-Dimensional Compressible Navier–Stokes Equations by a Generalized Implicit Method. *J. Comp. Phys.* 21, 372–397 (1977).

Brooks, A.N. and T.J.R. Hughes. Streamline Upwind/Petrov Galerkin Formulations for Convection Dominated Flows with Particular Emphasis on the Incompressible Navier–Stokes Equations; *Comp. Meth. Appl. Mech. Eng.* 32, 199–259 (1982).

Buning, P.G., I.T. Chiu, S. Obayashi, Y.M. Rizk and J.L. Steger. Numerical Simulation of the Integrated Space Shuttle Vehicle in Ascent; *AIAA*-88-4359-CP (1988).

Burstein, S.Z. and E. Rubin. Difference Methods for the Inviscid and Viscous Equations of a Compressible Gas; *J. Comp. Phys.* 2, 178 (1967).

Butcher, J.C. Implicit Runge–Kutta Processes; *Mathematics of Computation* 18(85), 50–64 (1964).

Butcher, J.C. *Numerical Methods for Ordinary Differential Equations;* John Wiley & Sons (2003).

Cabello, J., R. Löhner and O.-P. Jacquotte. A Variational Method for the Optimization of Two- and Three-Dimensional Unstructured Meshes; *AIAA*-92-0450 (1992).

Camelli, F. and R. Löhner. Assessing Maximum Possible Damage for Contaminant Release Events; *Engineering Computations* 21(7), 748–760 (2004).

Camelli, F.E. and R. Löhner. VLES Study of Flow and Dispersion Patterns in Heterogeneous Urban Areas; *AIAA*-06-1419 (2006).

Camelli, F.F., O. Soto, R. Löhner, W. Sandberg and R. Ramamurti. Topside LPD17 Flow and Temperature Study with an Implicit monolithic Scheme; *AIAA*-03-0969 (2003).

Carcaillet, R., S.R. Kennon and G.S. Dulikravitch. Generation of Solution-Adaptive Computational Grids Using Optimization; *Comp. Meth. Appl. Mech. Eng.* 57, 279–295 (1986).

Carey, G.F. *Grid Generation, Refinement, and Redistribution;* John Wiley & Sons (1993).

Carey, G.F. *Computational Grids: Generation, Adaption, and Solution;* Taylor & Francis (1997).

Cash, J.R. Review Paper. Efficient Numerical Methods for the Solution of Stiff Initial-Value Problems and Differential Algebraic Equations; *Proc. R. Soc. Lond. A: Mathematical, Physical and Engineering Sciences* 459, 2032, 797–815 (2003).

Cavendish, J.C. Automatic Triangulation of Arbitrary Planar Domains for the Finite Element Method; *Int. J. Num. Meth. Eng.* 8, 679–696 (1974).

Cavendish, J.C., D.A. Field and W.H. Frey. An Approach to Automatic Three-Dimensional Finite Element Generation; *Int. J. Num. Meth. Eng.* 21, 329–347 (1985).

Cebral, J.R. and R. Löhner. Conservative Load Projection and Tracking for Fluid-Structure Problems; *AIAA J.* 35(4), 687–692 (1997).

Cebral, J.R. and R. Löhner. From Medical Images to CFD Meshes; *Proc. 8th Int. Meshing Roundtable*, South Lake Tahoe, October (1999).

Cebral, J.R. and R. Löhner. From Medical Images to Anatomically Accurate Finite Element Grids; *Int. J. Num. Meth. Eng.* 51, 985–1008 (2001).

Cebral, J.R. and R. Löhner. Efficient Simulation of Blood Flow Past Complex Endovascular Devices Using an Adaptive Embedding Technique; *IEEE Transactions on Medical Imaging* 24(4), 468–476 (2005).

Cebral, J.R., R. Löhner, P.L. Choyke and P.J. Yim. Merging of Intersecting Triangulations for Finite Element Modeling; *J. of Biomechanics* 34, 815–819 (2001).

Cebral, J.R., F.E. Camelli and R. Löhner. A Feature-Preserving Volumetric Technique to Merge Surface Triangulations; *Int. J. Num. Meth. Eng.* 55, 177–190 (2002).

Chakravarthy, S.R. and K.Y. Szema. Euler Solver for Three-Dimensional Supersonic Flows with Subsonic Pockets; *J. Aircraft* 24(2), 73–83 (1987).

Chen, G. and C. Kharif. Two-Dimensional Navier–Stokes Simulation of Breaking Waves; *Physics of Fluids*, 11(1), 121–133 (1999).

Chiandussi, G., G. Bugeda and E. Oñate. A Simple Method for Automatic Update of Finite Element Meshes; *Comm. Num. Meth. Eng.* 16, 1–19 (2000).

Cho, Y., S. Boluriaan and P. Morris. Immersed Boundary Method for Viscous Flow Around Moving Bodies; *AIAA*-06-1089 (2006).

Choi, B.K., H.Y. Chin., Y.I. Loon and J.W. Lee. Triangulation of Scattered Data in 3D Space; *Comp. Aided Geom. Des.* 20, 239–248 (1988).

Chorin, A.J. A Numerical Solution for Solving Incompressible Viscous Flow Problems; *J. Comp. Phys.* 2, 12–26 (1967).

Chorin, A.J. Numerical Solution of the Navier–Stokes Equations; *Math. Comp.* 22, 745–762 (1968).

Clarke, D.K., H.A. Hassan and M.D. Salas. Euler Calculations for Multielement Airfoils Using Cartesian Grids; *AIAA*-85-0291 (1985).

Cleary. P.W. Discrete Element Modeling of Industrial Granular Flow Applications; *TASK. Quarterly - Scientific Bulletin* 2, 385–416 (1998).

Codina, R. Pressure Stability in Fractional Step Finite Element Methods for Incompressible Flows; *J. Comp. Phys.* 170, 112–140 (2001).

Codina, R. and O. Soto. A Numerical Model to Track Two-Fluid Interfaces Based on a Stabilized Finite Element Method and the Level Set Technique; *Int. J. Num. Meth. Fluids* 4, 293–301 (2002).

Colella, P. Multidimensional Upwind Methods for Hyperbolic Conservation Laws; *J. Comp. Phys.* 87, 171–200 (1990).

Collatz, L. *The Numerical Treatment of Differential Equations;* Springer-Verlag (1966).

Cook, B.K. and R.P. Jensen (eds.). *Discrete Element Methods;* ASCE (2002).

Coppola-Owen, A.H. and R. Codina. Improving Eulerian Two-Phase Flow Finite Element Approximation With Discontinuous Gradient Pressure Shape Functions; *Int. J. Num. Meth. Fluids* 49(12), 1287–1304 (2005).

Cortez, R. and M. Minion. The Blop Projection Method for Immersed Booundary Problems; *J. Comp. Phys.* 161, 428–453 (2000).

Coutinho, A.L.G.A., M.A.D. Martins, J.L.D. Alves, L. Landau and A. Morais. Edge-Based Finite Element Techniques for Non-Linear Solid Mechanics Problems; *Int. J. Num. Meth. Eng.* 50(9), 2053–2068 (2001).

Cowles, G. and L. Martinelli. Fully Nonlinear Hydrodynamic Calculations for Ship Design on Parallel Computing Platforms; *Proc. 21st Symp. on Naval Hydrodynamics*, Trondheim, Norway, June (1996).

Crispin, Y. Aircraft Conceptual Optimization Using Simulated Evolution; *AIAA*-94-0092 (1994).

Crumpton, P.I. and M.B. Giles. Implicit Time Accurate Solutions on Unstructured Dynamic Grids; *AIAA*-95-1671 (1995).

Cundall, P.A. Formulation of a Three-Dimensional Distinct Element Model - Part I: A Scheme to Detect and Represent Contacts in a System Composed of Many Polyhedral Blocks; *Int. J. Rock Mech. Min. Sci.* 25, 107–116 (1988).

Cundall, P.A. and O.D.L. Stack. A Discrete Numerical Model for Granular Assemblies; *Geotechnique* 29(1), 47–65 (1979).

Cuthill, E. and J. McKee. Reducing the Bandwidth of Sparse Symmetric Matrices; *Proc. ACM Nat. Conf.*, New York 1969, 157–172 (1969).

Cybenko, G. Dynamic Load Balancing of Distributed Memory Multiprocessors; *J. Parallel and Distr. Comp.* 7, 279–301 (1989).

Dadone, A. and B. Grossman. Progressive Optimization of Inverse Fluid Dynamic Design Problems; *Computers & Fluids* 29, 1–32 (2000).

Dadone, A. and B. Grossman. An Immersed Boundary Methodology for Inviscid Flows on Cartesian Grids; *AIAA*-02-1059 (2002).

Dadone, A. and B. Grossman. Fast Convergence of Inviscid Fluid Dynamic Design Problems; *Computers & Fluids* 32, 607–627 (2003).

Dadone, A., M. Valorani and B. Grossman. Smoothed Sensitivity Equation Method for Fluid Dynamic Design Problems; *AIAA J.* 38, 607–627 (2000).

Dahl, E.D. Mapping and Compiled Communication on the Connection Machine; *Proc. Distributed Memory Computer Conf.*, Charleston, SC, April (1990).

Dahlquist, G. A Special Stability Problem for Linear Multistep Methods; *BIT* 3, 27–43 (1963).

Dannenhoffer, J.F. and J.R. Baron. Robust Grid Adaptation for Complex Transonic Flows; *AIAA*-86-0495 (1986).

Darve, E. and R. Löhner. Advanced Structured-Unstructured Solver for Electromagnetic Scattering from Multimaterial Objects; *AIAA*-97-0863 (1997).

Davis, G.A. and O.O. Bendiksen. Unsteady Transonic Two-Dimensional Euler Solutions Using Finite Elements; *AIAA J.* 31, 1051–1059 (1993).

Davis, R.L. and J.F. Dannenhoffer. Adaptive Grid Embedding Navier–Stokes Technique for Cascade Flows; *AIAA*-89-0204 (1989).

Davis, R.L. and J.F. Dannenhoffer. 3-D Adaptive Grid-Embedding Euler Technique; *AIAA*-93-0330 (1993).

Dawes, W.N. Building Blocks Towards VR-Based Flow Sculpting; *AIAA*-05-1156 (2005).

Dawes, W.N. Towards a Fully Integrated Parallel Geometry Kernel, Mesh Generator, Flow Solver and Post-Processor; *AIAA*-06-0942 (2006).

De Jong, K. *Evolutionary Computing;* MIT Press (2006).

de Zeeuw, D. and K. Powell. An Adaptively-Refined Cartesian Mesh Solver for the Euler Equations; *AIAA*-91-1542 (1991).

Deb, K. *Multi-Objective Optimization Using Evolutionary Algorithms;* John Wiley & Sons (2001).

deFainchtein, R, S.T. Zalesak, R. Löhner and D.S. Spicer. Finite Element Simulation of a Turbulent MHD System: Comparison to a Pseudo-Spectral Simulation; *Comp. Phys. Comm.* 86, 25–39 (1995).

Degand, C. and C. Farhat. A Three-Dimensional Torsional Spring Analogy Method for Unstructured Dynamic Meshes; *Comp. Struct.* 80, 305–316 (2002).

Dekoninck, W. and T. Barth (eds.). *AGARD Rep.*787, *Proc. Special Course on Unstructured Grid Methods for Advection Dominated Flows*, VKI, Belgium, May (1992), Ch. 8.

del Pino, S. and O. Pironneau. Fictitious Domain Methods and Freefem3d; *Proc. ECCOMAS CFD Conf.*, Swansea, Wales (2001).

Deng, J., X.-M. Shao and A.-L. Ren. A New Modification of the Immersed-Boundary Method for Simulating Flows with Complex Moving Boundaries; *Int. J. Num. Meth. Fluids* 52, 1195–1213 (2006).

Devloo, P., J.T. Oden and P. Pattani. An H-P Adaptive Finite Element Method for the Numerical Simulation of Compressible Flows; *Comp. Meth. Appl. Mech. Eng.* 70, 203–235 (1988).

DeZeeuw, D. and K.G. Powell. An Adaptively Refined Cartesian Mesh Solver for the Euler Equations; *J. Comp. Phys.* 104, 56–68 (1993).

Diaz, A.R., N. Kikuchi and J.E. Taylor. A Method for Grid Optimization for the Finite Element Method; *Comp. Meth. Appl. Mech. Eng.* 41, 29–45 (1983).

Donea, J. A Taylor Galerkin Method for Convective Transport Problems; *Int. J. Num. Meth. Engng.* 20, 101–119 (1984).

Donea, J. and A. Huerta. *Finite Element Methods for Flow Problems;* John Wiley & Sons (2002).

Donea, J., S. Giuliani, H. Laval and L. Quartapelle. Solution of the Unsteady Navier–Stokes Equations by a Fractional Step Method; *Comp. Meth. Appl. Mech. Eng.* 30, 53–73 (1982).

Doorly, D. Parallel Genetic Algorithms for Optimization in CFD; pp. 251–270 in *Genetic Algorithms in Engineering and Computer Science* (G. Winter, J. Periaux, M. Galan and P. Cuesta eds.), John Wiley & Sons (1995).

Dougherty, F.C. and J. Kuan. Transonic Store Separation Using a Three-Dimensional Chimera Grid Scheme; *AIAA*-89-0637 (1989).

Drela, M. Pros and Cons of Airfoil Optimization; in *Frontiers of CFD'98* (D.A. Caughey and M.M. Hafez eds.) World Scientific (1998).

Dreyer, J.J. and L. Matinelli. Hydrodynamic Shape Optimization of Propulsor Configurations Using a Continuous Adjoint Approach; *AIAA*-01-2580 (2001).

Duarte, C.A. and J.T. Oden. H_p Clouds - A Meshless Method to Solve Boundary-Value Problems; *TICAM-Rep.* 95-05 (1995).

Duff, I.S. and G.A. Meurant. The Effect of Ordering on Preconditioned Conjugate Gradients; *BIT* 29, 635–657 (1989).

Dunavant, D.A. and B.A. Szabo. A Posteriori Error Indicators for the P-Version of the Finite Element Method; *Int. J. Num. Meth. Eng.* 19, 1851–1870 (1983).

Duncan, J. H. The Breaking and Non-Breaking Wave Resistance of a Two-Dimensional Hydrofoil; *J. Fluid Mesh.* 126, 507–516 (1983).

Dutto, L.C., W.G. Habashi, M.P. Robichaud and M. Fortin. A Method for Finite Element Parallel Viscous Compressible Flow Calculations; *Int. J. Num. Meth. Fluids* 19, 275–294 (1994).

Eiseman, P.R. *GridPro/az3000 Users Manual;* Program Development Corporation, White Plains, NY (1996).

Elliott, J. and J. Peraire. Aerodynamic Optimization on Unstructured Meshes with Viscous Effects; *AIAA*-97-1849 (1997).

Elliott, J. and J. Peraire. Constrained, Multipoint Shape Optimization for Complex 3-D Configurations; *Aeronautical J.* 102, 365–376 (1998).

Enright, D., D. Nguyen, F. Gibou and R. Fedkiw. Using the Particle Level Set Method and a Second Order Accurate Pressure Boundary Condition for Free Surface Flows; pp. 1–6 in *Proc. 4th ASME-JSME Joint Fluids Eng. Conf.* FEDSM2003-45144 (M. Kawahashi, A. Ogut and Y. Tsuji eds.), Honolulu, HI (2003).

Fadlun, E.A., R. Verzicco, P. Orlandi, J. Mohd-Yusof. Combined Immersed-Boundary Finite-Difference Methods for Three-Dimensional Complex Flow Simulations; *J. Comp. Phys.* 161, 35–60 (2000).

Faltisen, O.M. A Nonlinear Theory of Sloshing in Rectangular Tanks; *J. of Ship Research*, 18(4), 224–241 (1974).

Farhat, C., C. Degand, B. Koobus and M. Lesoinne. Torsional Springs for Two-Dimensional Dynamic Unstructured Fluid Meshes; *Comp. Meth. App. Mech. Eng.* 163, 231–245 (1998).

Farin, G. *Curves and Surfaces for Computer Aided Geometric Design;* Academinc Press (1990).

Farmer, J. R., L. Martinelli and A. Jameson. A Fast Multigrid Method for Solving Incompressible Hydrodynamic Problems With Free Surfaces; *AIAA J.* 32(6), 1175–1182 (1993).

Fedorenko, R.P. A Relaxation Method for Solving Elliptic Difference Equations; *USSR Comp. Math. Math. Phys.* 1(5), 1092–1096 (1962).

Fedorenko, R.P. The Speed of Convergence of an Iterative Process; *USSR Comp. Math. Math. Phys.* 4(3), 227–235 (1964).

Fekken, G., A.E.P. Veldman and B. Buchner. Simulation of Green Water Loading Using the Navier–Stokes Equations; *Proc. 7th Int. Conf. on Num. Ship Hydrodynamics*, Nantes, France (1999).

Feng, Y.T., K. Han and D.R.J. Owen. An Advancing Front Packing of Polygons, Ellipses and Spheres; pp. 93–98 in *Discrete Element Methods* (B.K. Cook and R.P. Jensen eds.); ASCE (2002).

Ferziger, J.H. and M. Peric. *Computational Methods for Fluid Dynamics;* Springer-Verlag (1999).

Fischer, P.F. Projection Techniques for Iterative Solution of $Ax = b$ With Successive Right-Hand Sides; *Comp. Meth. Appl. Mech. Eng.* 163, 193–204 (1998).

Flower, J., S. Otto and M. Salama. Optimal Mapping of Irregular Finite Element Domains to Parallel Processors; 239–250 (1990).

Floyd, R.W. Treesort 3; *Comm. ACM* 7, 701 (1964).

Fortin, M. and F. Thomasset. Mixed Finite Element Methods for Incompressible Flow Problems; *J. Comp. Phys.* 31, 113–145 (1979).

Franca, L.P. and S.L. Frey. Stabilized Finite Element Methods: II. The incompressible Navier–Stokes Equations; *Comp. Meth. Appl. Mech. Eng.* 99, 209–233 (1992).

Franca, L.P., T.J.R. Hughes, A.F.D. Loula and I. Miranda. A New Family of Stable Elements for the Stokes Problem Based on a Mixed Galerkin/Least-Squares Finite Element Formulation; pp. 1067–1074 in *Proc. 7th Int. Conf. Finite Elements in Flow Problems* (T.J. Chung and G. Karr eds.), Huntsville, AL (1989).

Freitag, L.A. and C.-O. Gooch. Tetrahedral Mesh Improvement Using Swapping and Smoothing; *Int. J. Num. Meth. Eng.*, 40, 3979–4002 (1997).

Frey, W.H. Selective Refinement: A New Strategy for Automatic Node Placement in Graded Triangular Meshes; *Int. J. Num. Meth. Eng.* 24, 2183–2200 (1987).

Frey, P.J. About Surface Remeshing; pp. 123–136 in *Proc. 9th Int. Meshing Roundtable* (2000).

Frey, P.J. and H. Borouchaki. Geometric Surface Mesh Optimization; *Computing and Visualization in Science* 1, 113–121 (1998).

Frey, P.J. and P.L. George. *Mesh Generation Application to Finite Elements;* Hermes Science Publishing, Oxford, Paris (2000).

Fry, M.A. and D.L. Book. Adaptation of Flux-Corrected Transport Codes for Modelling Dusty Flows; *Proc. 14th Int.Symp. on Shock Tubes and Waves* (R.D. Archer and B.E. Milton eds.); New South Wales University Press (1983).

Frykestig, J. Advancing Front Mesh Generation Techniques with Application to the Finite Element Method; *Pub. 94:10;* Chalmers University of Technology, Göteborg, Sweden (1994).

Fuchs, A. Automatic Grid Generation with Almost Regular Delaunay Tetrahedra; pp. 133–148 in *Proc. 7th Int. Meshing Roundtable* (1998).

Fursenko, A. Unstructured Euler Solvers and Underlying Algorithms for Strongly Unsteady Flows; *Proc. 5th Int. CFD Symp.*, Sendai (1993).

Fyfe, D.E., J.H. Gardner, M. Picone and M.A. Fry. Fast Three-Dimensional Flux-Corrected Transport Code for Highly Resolved Compressible Flow Calculations; *Springer Lecture Notes in Physics* 218, 230–234, Springer-Verlag (1985).

Gage, P. and I. Kroo. A Role for Genetic Algorithms in a Preliminary Design Environment; *AIAA*-93-3933 (1993).

Gaski, J. and R.L. Collins. SINDA 1987-ANSI Revised User's Manual. Network Analysis Associates, Inc. (1987).

Gen, M. and R. Cheng. *Genetic Algorithms and Engineering Optimization;* John Wiley & Sons (1997).

Gentzsch, W. Über ein verbessertes explizites Einschrittverfahren zur Lösung parabolischer Differentialgleichungen; *DFVLR-Report* (1980).

Gentzsch, W. and A. Schlüter. Über ein Einschrittverfahren mit zyklischer Schrittweitenänderung zur Lösung parabolischer Differentialgleichungen; *ZAMM* 58, T415–T416 (1978).

George, P.L. *Automatic Mesh Generation;* John Wiley & Sons (1991).

George, P.L. and H. Borouchaki. *Delaunay Triangulation and Meshing;* Editions Hermes, Paris (1998).

George, P.L. and F. Hermeline. Delaunay's Mesh of a Convex Polyhedron in Dimension d. Application to Arbitrary Polyhedra; *Int. J. Num. Meth. Eng.* 33, 975–995 (1992).

George, A. and J.W. Liu. *Computer Solution of Large Sparse Positive Definite Systems;* Prentice-Hall (1981).

George, P.L., F. Hecht and E. Saltel. Fully Automatic Mesh Generation for 3D Domains of any Shape; *Impact of Computing in Science and Engineering* 2(3), 187–218 (1990).

George, P.L., F. Hecht and E. Saltel. Automatic Mesh Generator With Specified Boundary; *Comp. Meth. Appl. Mech. Eng.* 92, 269–288 (1991).

Giannakoglou, K. Design of Optimal Aerodynamic Shapes Using Stochastic Optimization Methods and Computational Intelligence; *Progress in Aerospace Sciences* 38, 43–76 (2002).

Giles, M. and E. Süli. Adjoints Methods for PDEs: A Posteriori Error Analysis and Postprocessing by Duality; pp. 145–236 in: A. Iserles (Ed.), *Acta Numerica 2002*, Cambridge University Press (2002).

Giles, M., M. Drela and W. Thompkins. Newton Solution of Direct and Inverse Transonic Euler Equations; *AIAA*-85-1530-CP (1985).

Gilmanov, A. and F. Sotiropoulos. A Hybrid Cartesian/Immersed Boundary Method for Simulating Flows with 3-D, Geometrically Complex Moving Objects; *J. Comp. Phys.* 207, 457–492 (2005).

Gilmanov, A., F. Sotiropoulos and E. Balaras. A General Reconstruction Algorithm for Simulating Flows with Complex 3D Immersed Boundaries on Cartesian Grids; *J. Comp. Phys.* 191(2), 660–669 (2003).

Gingold, R.A. and J.J. Monahghan. Shock Simulation by the Particle Method SPH; *J. Comp. Phys.* 52, 374–389 (1983).

Glowinski, R., T.W. Pan and J. Periaux. A Fictitious Domain Method for External Incompressible Flow Modeled by the Navier–Stokes Equations; *Comp. Meth. Appl. Mech. Eng.* 112(1–4), 133–148 (1994).

Glowinski, R., T.W. Pan, T.I. Hesla and D.D. Joseph. A Distributed Lagrange Multiplier/ Fictitious Domain Method for Particulate Flows; *Int. J. Multiphase Flow* 25(5), 755–794 (1999).

Gnoffo, P.A. A Finite-Volume, Adaptive Grid Algorithm Applied to Planetary Entry Flowfields; *AIAA J.* 21, 1249–1254 (1983).

Godunov, S.K. Finite Difference Method for Numerical Computation of Discontinuous Solutions of the Equations of Fluid Dynamics; *Mat. Sb.* 47, 271–306 (1959).

Goldberg, D.E. *Genetic Algorithms in Search, Optimization and Machine Learning;* Addison-Wesley (1989).

Goldstein, D., R. Handler and L. Sirovich. Modeling a No-Slip Flow Boundary with an External Force Field; *J. Comp. Phys.* 105, 354–366 (1993).

Gregoire, J.P., J.P. Benque, P. Lasbleiz and J. Goussebaile. 3-D Industrial Flow Calculations by Finite Element Method; *Springer Lecture Notes in Physics* 218, 245–249 (1985).

Gresho, P.M. and S.T. Chan. On the Theory of Semi-Implicit Projection Methods for Viscous Incompressible Flows and its Implementation via a Finite Element Method That Introduces a Nearly-Consistent Mass Matrix; *Int. J. Num. Meth. Fluids* 11, 587–620 (1990).

Gresho, P.M. and R.L. Sani. *Incompressible Flow and the Finite Element Method; Vol. 1: Advection-Diffusion, Vol. 2: Isothermal Laminar Flow;* John Wiley & Sons (2000).

Gresho, P.M., C.D. Upson, S.T. Chan and R.L. Lee. Recent Progress in the Solution of the Time-Dependent, Three-Dimensional, Incompressible Navier–Stokes Equations; pp. 153–162 in *Proc. 4th Int. Symp. Finite Element Methods in Flow Problems* (T. Kawai ed.), University of Tokio Press (1982).

Griewank, A. and G.F. Corliss (eds.). *Automatic Differentiation of Algorithms: Theory, Implementation and Applications;* SIAM, Philadelphia (1991).

Groves, N., T.T. Huang and M.S. Chang. Geometric Characteristics of DARPA SUBOFF Models; *DTRC/SHD*-1298-01 (1998).

Guest, J.K. and J.H. Prévost. Topology Optimization of Creeping Flows Using a Darcy-Stokes Finite Element; *Int. J. Num. Meth. Eng.* 66(3), 461–484 (2006).

Gunzburger, M.D. Mathematical Aspects of Finite Element Methods for Incompressible Viscous Flows; pp. 124–150 in *Finite Elements: Theory and Application* (D.L. Dwoyer, M.Y. Hussaini and R.G. Voigt eds.), Springer-Verlag (1987).

Gunzburger, M.D. and R. Nicolaides (eds.). *Incompressible Computational Fluid Dynamics: Trends and Advances;* Cambridge University Press (1993).

Guruswamy, G.P. and C. Byun. Fluid-Structural Interactions Using Navier–Stokes Flow Equations Coupled with Shell Finite Element Structures; *AIAA*-93-3087 (1993).

Hackbusch, W. and U. Trottenberg (eds.). Multigrid Methods; *Lecture Notes in Mathematics* 960, Springer-Verlag (1982).

Hafez, M.M. (ed.). *Numerical Simulations of Incompressible Flows;* World Scientific (2003).

Hafez, M., E. Parlette and M. Salas. Convergence Acceleration of Iterative Solutions of Euler Equations for Transonic Flow Computations; *AIAA*-85-1641 (1985).

Haftka, R.T. Sensitivity Calculations for Iteratively Solved Problems; *Int. J. Num. Meth. Eng.* 21, 1535–1546 (1985).

Hagen, T.R., K.-A. Lie and J.R. Natvig. Solving the Euler Equations on Graphics Processing Units; *Proc. ICCS06* (2006).

Hairer, E. and G. Wanner. Algebraically Stable and Implementable Runge–Kutta Methods of High Order; *SIAM J. Num. Analysis* 18(6), 1098–1108 (1981).

Hanna, S.R., M.J. Brown, F.E. Camelli, S.T. Chan, W.J. Coirier, O.R. Hansen, A.H. Huber, S. Kim and R.M. Reynolds. Detailed Simulations of Atmospheric Flow and Dispersion in Downtown Manhattan: An Application of Five Computational Fluid Dynamics Models; *Bull. Am. Meteorological Soc.* December 2006, 1713–1726 (2006).

Hansbo, P. The Characteristic Streamline Diffusion Method for the Time-Dependent Incompressible Navier–Stokes Equations; *Comp. Meth. Appl. Mech. Eng.* 99, 171–186 (1992).

Harten, A. High Resolution Schemes for Hyperbolic Conservation Laws; *J. Comp. Phys.* 49, 357–393 (1983).

Harten, A. and G. Zwaas. Self-Adjusting Hybrid Schemes for Shock Computations; *J. Comp. Phys.* 6, 568–583 (1972).

Hassan, O., K. Morgan and J. Peraire. An Implicit Finite Element Method for High Speed Flows; *AIAA*-90-0402 (1990).

Hassan, O., E.J. Probert, K. Morgan. Unstructured Mesh Procedures for the Simulation of Three-Dimensional Transient Compressible Inviscid Flows with Moving Boundary Components; *Int. J. Num. Meth. Fluids* 27(1–4), 41–55 (1998).

Hassine, M., S. Jan and M. Masmoudi. From Differential Calculus to 0-1 Optimization; *ECCOMAS 2004*, Jyväskylä, Finland (2004).

Haug, E., H. Charlier, J. Clinckemaillie, E. DiPasquale, O. Fort, D. Lasry, G. Milcent, X. Ni, A.K. Pickett and R. Hoffmann. Recent Trends and Developments of Crashworthiness Simulation Methodologies and their Integration into the Industrial Vehicle Design Cycle; *Proc. Third European Cars/Trucks Simulation Symposium (ASIMUTH)*, October 28–30 (1991).

Hay, A. and M. Visonneau. Computation of Free-Surface Flows with Local Mesh Adaptation; *Int. J. Num. Meth. Fluids* 49, 785–816 (2005).

Hestenes, M. and E. Stiefel. Methods of Conjugate Gradients for Solving Linear Systems. *J. Res. Nat Bur. Standards* 49, 409–436 (1952).

Hétu, J.-F. and D.H. Pelletier. Adaptive Remeshing for Viscous Incompressible Flows; *AIAA J.* 30(8), 1986–1992 (1992).

Hicks, R.M. and P.A. Henne. Wing Design by Numerical Optimization; *J. of Aircraft* 15, 407–412 (1978).

Hino, T. Computation of Free Surface Flow Around an Advancing Ship by the Navier–Stokes Equations; *Proc. 5th Int. Conf. Num. Ship Hydrodynamics*, Hiroshima, Japan (1989).

Hino, T. An Unstructured Grid Method for Incompressible Viscous Flows with a Free Surface; *AIAA*-97-0862 (1997).

Hino, T. Shape Optimization of Practical Ship Hull Forms Using Navier–Stokes Analysis; *Proc. 7th Int. Conf. Num. Ship Hydrodynamics*, Nantes, France, July (1999).

Hino, T., L. Martinelli and A. Jameson. A Finite-Volume Method with Unstructured Grid for Free Surface Flow; *Proc. 6th Int. Conf. on Numerical Ship Hydrodynamics*, Iowa City, August (1993).

Hirsch, C. *Numerical Computation of Internal and External Flow;* John Wiley & Sons (1991).

Hirt, C.W. and B.D. Nichols. Volume of Fluid (VOF) Method for the Dynamics of Free Boundaries; *J. Comp. Phys.* 39, 201–225 (1981).

Hoffmann, K.A. and S.T. Chiang. *Computational Fluid Dynamics;* Engineering Education System (1998).

Holmes, D.G. and S.C. Lamson. Compressible Flow Solutions on Adaptive Triangular Meshes; Open Forum *AIAA, Reno'86, Meeting* (1986).

Holmes, D.G. and D.D. Snyder. The Generation of Unstructured Triangular Meshes Using Delaunay Triangulation; pp. 643–652 in *Numerical Grid Generation in Computational Fluid Dynamics* (Sengupta *et al.* eds.); Pineridge Press, Swansea, Wales (1988).

Hoppe, H., T. DeRose, T. Duchamp, J. McDonald and W. Stuetzle. Surface Reconstruction from Unorganized Points; *Comp. Graph.* 26(2), 71–78 (1992).

Hoppe, H., T. DeRose, T. Duchamp, J. McDonald and W. Stuetzle. Mesh Optimization; pp. 19–26 in *Proc. Comp. Graph. Ann. Conf.* (1993).

Hou, G.J.W., V. Maroju, A.C. Taylor, V.M. Korivi and P.A. Newman. Transonic Turbulent Airfoil Design Optimization with Automatic Differentiation in Incremental Iterative Form; *AIAA*-95-1692-CP (1995).

Huang, C.J. and S.J. Wu. Global and Local Remeshing Algorithm for Compressible Flows; *J. Comp. Phys.* 102, 98–113 (1992).

Huang, T.T., H.L. Liu, N.C. Groves, T.J. Forlini, J.N. Blanton and S. Gowing. Measurements of Flows over an Axisymmetric Body with Various Appendages (DARPA SUBOFF Experiments); *Proc. 19th Symp. Naval Hydrodynamics*, Seoul, Korea (1992).

Huet, F. Generation de Maillage Automatique dans les Configurations Tridimensionelles Complexes Utilization d'une Methode de 'Front'; *AGARD-CP*-464, 17 (1990).

Huffenus, J.D. and D. Khaletzky. A Finite Element Method to Solve the Navier–Stokes Equations Using the Method of Characteristics; *Int. J. Num. Meth. Fluids* 4, 247–269 (1984).

Hughes, T.J.R. and T.E. Tezduyar. Finite Element Methods for First-Order Hyperbolic Systems with Particular Emphasis on the Compressible Euler Equations; *Comp. Meth. Appl. Mech. Eng.* 45, 217–284 (1984).

Hughes, T.J.R., M. Levit and J. Winget. Element-by-Element Implicit Algorithms for Heat Conduction; *J. Eng. Mech.* 109(2) (1983a).

Hughes, T.J.R., M. Levit and J. Winget. An Element-by-Element Solution Algorithm for Problems in Structural and Solid Mechanics; *Comp. Meth. Appl. Mech. Eng.* 36, 241–254 (1983b).

Hughes, T.J.R., J. Winget, M. Levit and T.E. Tezduyar. New Alternating Direction Procedures in Finite Element Analysis Based Upon EBE Approximate Factorizations; pp. 75–109 in *Computer Methods for Nonlinear Solids and Structural Mechanics*, AMD-Vol. 54, ASME, New York (1983c).

Huijsmans, R.H.M. and E. van Grosen. Coupling Freak Wave Effects with Green Water Simulations; *Proc. of the 14th ISOPE*, Toulon, France, May 23–28 (2004).

Huyse, L. and R.M. Lewis. Aerodynamic Shape Optimization of Two-Dimensional Airfoils Under Uncertain Conditions; *NASA/CR*-2001-210648, *ICASE Report No. 2001-1* (2001).

Hystopolulos, E. and R.L. Simpson. Critical Evaluation of Recent Second-Order Closure Models; *AIAA*-93-0081 (1993).

Idelsohn, S., E. Oñate, N. Calvo and F. Del Pin. The Meshless Finite Element Method; *Int. J. Num. Meth. Eng.* 58(6), 893–912 (2003).

Ito, Y. and K. Nakahashi. Direct Surface Triangulation Using Stereolithography Data; *AIAA J.* 40(2), 490–496 (2002).

ITTC. Cooperative Experiments on Series-60 Hull at UT-Tank; *17th ITTC Resistance Committee Report* (2nd edn) (1983a).

ITTC. Cooperative Experiments on Wigley Parabolic Models in Japan; *17th ITTC Resistance Committee Report* (2nd edn) (1983b).

Jacquotte, O.-P. and J. Cabello. Three-Dimensional Grid Generation Method Based on a Variational Principle; *Rech. Aerosp.* 1990-4, 8–19 (1990).

Jakiela, M.J., C.D. Chapman, J. Duda, A. Adewuya and K. Saitou. Continuum Structural Topology Design with Genetic Algorithms; *Comp. Meth. Appl. Mech. Eng.* 186, 339–356 (2000).

Jameson, A. Acceleration of Transonic Potential Flow Calculations on Arbitrary Meshes by the Multi-Grid Method; *AIAA*-79-1458-CP (1979).

Jameson, A. Solution of the Euler Equations by a Multigrid Method; *Appl. Math. Comp.* 13, 327–356 (1983).

Jameson, A. Aerodynamic Design via Control Theory; *J. Scientific Computing* 3, 233–260 (1988).

Jameson, A. Optimum Aerodynamic Design Using Control Theory; in *CFD Review 1995*, John Wiley & Sons (1995).

Jameson, A. and D. Caughey. A Finite Volume Method for Transonic Potential Flow Calculations; pp. 35–54 in *Proc. AIAA 3rd CFD Conf.*, Albuquerque, NM (1977).

Jameson, A., W. Schmidt and E. Turkel. Numerical Solution of the Euler Equations by Finite Volume Methods using Runge–Kutta Time-Stepping Schemes; *AIAA*-81-1259 (1981).

Jameson, A. and S. Yoon. Multigrid Solution of the Euler Equations Using Implicit Schemes; *AIAA*-85-0293 (1985).

Jameson, A., T.J. Baker and N.P. Weatherhill. Calculation of Inviscid Transonic Flow over a Complete Aircraft; *AIAA*-86-0103 (1986).

Jespersen, D. and C. Levit. A Computational Fluid Dynamics Algorithm on a Massively Parallel Machine; *AIAA*-89-1936-CP (1989).

Jin, H. and R.I. Tanner. Generation of Unstructured Tetrahedral Meshes by the Advancing Front Technique; *Int. J. Num. Meth. Eng.* 36, 1805–1823 (1993).

Joe, B. Construction of Three-Dimensional Delaunay Triangulations Using Local Transformations; *Computer Aided Geometric Design* 8, 123–142 (1991a).

Joe, B. Delaunay Versus Max-Min Solid Angle Triangulations for Three-Dimensional Mesh Generation; *Int. J. Num. Meth. Eng.* 31, 987–997 (1991b).

Johnsson, C. and P. Hansbo. Adaptive Finite Element Methods in Computational Mechanics; *Comp. Meth. Appl. Mech. Eng.* 101, 143–181 (1992).

Johnson, A.A. and T.E. Tezduyar. Mesh Update Strategies in Parallel Finite Element Computations of Flow Problems With Moving Boundaries and Interfaces; *Comp. Meth. App. Mech. Eng.* 119, 73–94 (1994).

Johnson, A.A. and T.E. Tezduyar. 3D Simulation of Fluid-Particle Interactions with the Number of Particles Reaching 100; *Comp. Meth. App. Mech. Eng.* 145, 301–321 (1997).

Jothiprasad, G., D.J. Mavriplis and D.A. Caughey. Higher-Order Time Integration Schemes for the Unsteady Navier–Stokes Equations on Unstructured Meshes; *J. Comp. Phys.* 191(2), 542–566 (2003).

J. Comp. Phys., Vol. 48 (1982).

Jue, T.C., B. Ramaswamy and J.E. Akin. Finite Element Simulation of 2-D Benard Convection with Gravity Modulation; pp. 87–101 in FED-Vol. 123, ASME (1991).

Kallinderis, J.G. and J.R. Baron. Adaptation Methods for a New Navier–Stokes Algorithm; *AIAA*-87-1167-CP, Hawaii (1987).

Kallinderis, Y. and A. Chen. An Incompressible 3-D Navier–Stokes Method with Adaptive Hybrid Grids; *AIAA*-96-0293 (1996).

Kallinderis, Y. and S. Ward. Prismatic Grid Generation with an Efficient Algebraic Method for Aircraft Configurations; *AIAA*-92-2721 (1992).

Karbon, K.J. and S. Kumarasamy. Computational Aeroacoustics in Automotive Design, Computational Fluid and Solid Mechanics; *Proc. First MIT Conference on Computational Fluid and Solid Mechanics*, 871–875, Boston, June (2001).

Karbon, K.J. and R. Singh. Simulation and Design of Automobile Sunroof Buffeting Noise Control; *8th AIAA-CEAS Aero-Acoustics Conf.*, Brenckridge, June (2002).

Karman, S.L. SPLITFLOW: A 3-D Unstructured Cartesian/ Prismatic Grid CFD Code for Complex Geometries; *AIAA*-95-0343 (1995).

Kelly, D.W., S. Nakazawa, O.C. Zienkiewicz and J.C. Heinrich. A Note on Anisotropic Balancing Dissipation in Finite Element Approximation to Convection Diffusion Problems. *Int. J. Num. Meth. Eng.* 15, 1705–1711 (1980).

Kicinger, R., T. Arciszewski and K. De Jong. Evolutionary Computation and Structural Design: A Survey of the State-of-the-Art; *Comp. Struct.* 83, 1943–1978 (2005).

Kim, J. and P. Moin. Application of a Fractional-Step Method to Incompressible Navier–Stokes Equations; *J. Comp. Phys.* 59, 308–323 (1985).

Kim, J., D. Kim and H. Choi. An Immersed-Boundary Finite-Volume Method for Simulation of Flow in Complex Geometries; *J. Comp. Phys.* 171, 132–150 (2001).

Kirkpatrick, R.C. Nearest Neighbor Algorithm; pp. 302–309 in *Springer Lecture Notes in Physics* 238 (M.J. Fritts, W.P. Crowley and H. Trease eds.); Springer-Verlag (1985).

Knuth, D.E. *The Art of Computer Programming*, Vols. 1–3; Addison-Wesley, Reading, MA (1973).

Kölke, A. Modellierung und Diskretisierung bewegter Diskontinuitäten in Randgekoppelten Mehrfeldaufgaben; *PhD Thesis*, TU Braunschweig (2005).

Kumano,T., S. Jeong, S. Obayashi, Y. Ito, K. Hatanaka and H. Morino. Multidisciplinary Design Optimization of Wing Shape with Nacelle and Pylon; *Proc. ECCOMAS CFD 2006* (P. Wesseling, E. Oñate, J. Périaux eds.) (2006).

Kuruvila, G., S. Ta’asan and M.D. Salas. Airfoil Design and Optimization by the One-Shot Method; *AIAA*-95-0478 (1995).

Kutler, P., W.A. Reinhardt and R.F. Warming. Multishocked, Three-Dimensional Supersonic Flowfields with Real Gas Effects; *AIAA J.* 11(5), 657–664 (1973).

Kuzmin, D. Positive Finite Element Schemes Based on the Flux-Corrected Transport Procedure; pp. 887–888 in *Computational Fluid and Solid Mechanics*, Elsevier (2001).

Kuzmin, D. and S. Turek. Flux Correction Tools for Finite Elements; *J. Comp. Phys.* 175, 525–558 (2002).

Kuzmin, D., M. Möller and S. Turek. Multidimensional FEM-FCT Schemes for Arbitrary Time-Stepping; *Int. J. Num. Meth. Fluids* 42, 265–295 (2003).

Kuzmin, D., R. Löhner and S. Turek (eds.). *Flux-Corrected Transport;* Springer-Verlag (2005).

Lai, M.C. and C.S. Peskin. An Immersed Boundary Method With Formal Second- Order Accuracy and Reduced Numerical Viscosity; *J. Comp. Phys.* 160, 132–150 (2000).

Landrini, M., A. Colagorossi and O.M. Faltisen. Sloshing in 2-D Flows by the SPH Method; *Proc. 8th Int. Conf. Num. Ship Hydrodynamics*, Busan, Korea (2003).

Landsberg, A.M. and J.P. Boris. The Virtual Cell Embedding Method: A Simple Approach for Gridding Complex Geometries; *AIAA*-97-1982 (1997).

Lapidus, A. A Detached Shock Calculation by Second-Order Finite Differences; *J. Comp. Phys.* 2, 154–177 (1967).

Lawrence, S.L., D.S. Chaussee and J.C. Tannehill; Development of a Three-Dimensional Upwind Parabolized Navier–Stokes Code; *AIAA J.* 28(6), 971–972 (1991).

Lax, P.D. and B. Wendroff. Systems of Conservation Laws; *Comm. Pure Appl. Math.* 13, 217–237 (1960).

Le Beau, G.J. and T.E. Tezduyar. Finite Element Computation of Compressible Flows with the SUPG Formulation; *Advances in Finite Element Analysis in Fluid Dynamics* FED-Vol. 123, 21–27, ASME, New York (1991).

Lee, D.T. and B.J. Schachter. Two Algorithms for Constructing a Delaunay Triangulation; *Int. J. Comp. Inf. Sc.* 9(3), 219–242 (1980).

LeGresley, P.A. and J.J. Alonso. Airfoil Design Optimization Using Reduced Order Models Based on Proper Orthogonal Decomposition; *AIAA*-00-2545 (2000).

LeGresley, P., E. Elsen and E. Darve. Calculation of the Flow Over a Hypersonic Vehicle Using a GPU; *Proc. Supercomputing'07* (2007).

Lesoinne, M. and Ch. Farhat. Geometric Conservation Laws for Flow Problems With Moving Boundaries and Deformable Meshes, and Their Impact on Aeroelastic Computations; *Comp. Meth. Appl. Mech. Eng.* 134, 71–90 (1996).

LeVeque, R.J. and D. Calhoun. Cartesian Grid Methods for Fluid Flow in Complex Geometries; pp. 117–143 in *Computational Modeling in Biological Fluid Dynamics* (L.J. Fauci and S. Gueron, eds.), IMA Volumes in Mathematics and its Applications 124, Springer-Verlag (2001).

LeVeque, R.J. and Z. Li. The Immersed Interface Method for Elliptic Equations with Discontinuous Coefficients and Singular Sources; *SIAM J. Num. Anal.* 31, 1019–1044 (1994).

Li, W. L. Huyse and S. Padula. Robust Airfoil Optimization to Achieve Consistent Drag Reduction Over a Mach Range; *NASA/CR*-2001-211042, *ICASE Report No. 2001–22* (2001).

Li, Y., T. Kamioka, T. Nouzawa, T. Nakamura, Y. Okada and N. Ichikawa. Verification of Aerodynamic Noise Simulation by Modifying Automobile Front-Pillar Shape; *JSAE 20025351, JSAE Annual Conf.*, Tokyo, July (2002).

Liou, J. and T.E. Tezduyar. Clustered Element-by-Element Computations or Fluid Flow; Ch. 9 in *Parallel Computational Fluid Dynamics* (H.D. Simon ed.); MIT Press, Cambridge, MA (1992).

Liu, W.K., Y. Chen, S. Jun, J.S. Chen, T. Belytschko, C. Pan, R.A. Uras and C.T. Chang. Overview and Applications of the Reproducing Kernel Particle Methods; *Archives Comp. Meth. Eng.* 3(1), 3–80 (1996).

Lo, S.H. A New Mesh Generation Scheme for Arbitrary Planar Domains; *Int. J. Num. Meth. Eng.* 21, 1403–1426 (1985).

Lo, S.H. Finite Element Mesh Generation over Curved Surfaces; *Comp. Struc.* 29, 731–742 (1988).

Lo, S.H. Automatic Mesh Generation over Intersecting Surfaces; *Int. J. Num. Meth. Eng.* 38, 943–954 (1995).

Löhner, R. An Adaptive Finite Element Scheme for Transient Problems in CFD; *Comp. Meth. Appl. Mech. Eng.* 61, 323–338 (1987).

Löhner, R. Some Useful Data Structures for the Generation of Unstructured Grids; *Comm. Appl. Num. Meth.* 4, 123–135 (1988a).

Löhner, R. An Adaptive Finite Element Solver for Transient Problems with Moving Bodies; *Comp. Struct.* 30, 303–317 (1988b).

Löhner, R. Adaptive H-Refinement on 3-D Unstructured Grids for Transient Problems; *AIAA*-89-0365 (1989a).

Löhner, R. Adaptive Remeshing for Transient Problems; *Comp. Meth. Appl. Mech. Eng.* 75, 195–214 (1989b).

Löhner, R. A Fast Finite Element Solver for Incompressible Flows; *AIAA*-90-0398 (1990a).

Löhner, R. Three-Dimensional Fluid-Structure Interaction Using a Finite Element Solver and Adaptive Remeshing; *Computer Systems in Engineering* 1(2–4), 257–272 (1990b).

Löhner, R. The FEFLO-Family of CFD Codes; *Proc. 2nd Int. ESI PAM-Workshop*, Paris, November (1991).

Löhner, R. Matching Semi-Structured and Unstructured Grids for Navier–Stokes Calculations; *AIAA*-93-3348-CP (1993).

Löhner, R. Robust, Vectorized Search Algorithms for Interpolation on Unstructured Grids; *J. Comp. Phys.* 118, 380–387 (1995).

Löhner, R. Regridding Surface Triangulations; *J. Comp. Phys.* 126, 1–10 (1996).

Löhner, R. Computational Aspects of Space-Marching; *AIAA*-98-0617 (1998).

Löhner, R. Overlapping Unstructured Grids; *AIAA*-01-0439 (2001).

Löhner, R. Multistage Explicit Advective Prediction for Projection-Type Incompressible Flow Solvers; *J. Comp. Phys.* 195, 143–152 (2004).

Löhner, R. Projective Prediction of Pressure Increments; *Comm. Num. Meth. Eng.* 21(4), 201–207 (2005).

Löhner, R. and J. Ambrosiano. A Vectorized Particle Tracer for Unstructured Grids; *J. Comp. Phys.* 91(1), 22–31 (1990).

Löhner, R. and J.D. Baum. Numerical Simulation of Shock Interaction with Complex Geometry Three-Dimensional Structures using a New Adaptive H-Refinement Scheme on Unstructured Grids; *AIAA*-90-0700 (1990).

Löhner, R. and J.D. Baum. Three-Dimensional Store Separation Using a Finite Element Solver and Adaptive Remeshing; *AIAA*-91-0602 (1991).

Löhner, R. and J.D. Baum. Adaptive H-Refinement on 3-D Unstructured Grids for Transient Problems; *Int. J. Num. Meth. Fluids* 14, 1407–1419 (1992).

Löhner, R. and F. Camelli. Dynamic Deactivation for Advection-Dominated Contaminant Flow; *Comm. Num. Meth. Eng.* 20, 639–646 (2004).

Löhner, R. and F. Camelli. Optimal Placement of Sensors for Contaminant Detection Based on Detailed 3-D CFD Simulations; *Engineering Computations* 22(3), 260–273 (2005).

Löhner, R. and M. Galle. Minimization of Indirect Addressing for Edge-Based Field Solvers; *AIAA*-02-0967 (2002).

Löhner, R. and K. Morgan. An Unstructured Multigrid Method for Elliptic Problems; *Int. J. Num. Meth. Eng.* 24, 101–115 (1987).

Löhner, R. and E. Oñate. An Advancing Point Grid Generation Technique; *Comm. Num. Meth. Eng.* 14, 1097–1108 (1998).

Löhner, R. and E. Oñate. A General Advancing Front Technique for Filling Space With Arbitrary Objects; *Int. J. Num. Meth. Eng.* 61, 1977–1991 (2004).

Löhner, R. and P. Parikh. Three-Dimensional Grid Generation by the Advancing Front Method; *Int. J. Num. Meth. Fluids* 8, 1135–1149 (1988).

Löhner, R. and R. Ramamurti. A Load Balancing Algorithm for Unstructured Grids; *Comp. Fluid Dyn.* 5, 39–58 (1995).

Löhner, R. and C. Yang. Improved ALE Mesh Velocities for Moving Bodies; *Comm. Num. Meth. Eng.* 12, 599–608 (1996).

Löhner, R., K. Morgan and O.C. Zienkiewicz. The Solution of Nonlinear Systems of Hyperbolic Equations by the Finite Element Method; *Int. J. Num. Meth. Fluids* 4, 1043–1063 (1984).

Löhner, R., K. Morgan and J. Peraire. A Simple Extension to Multidimensional Problems of the Artificial Viscosity due to Lapidus; *Comm. Appl. Num. Meth.* 1, 141–147 (1985a).

Löhner, R., K. Morgan, J. Peraire and O.C. Zienkiewicz. Finite Element Methods for High Speed Flows; *AIAA*-85-1531-CP (1985b).

Löhner, R., K. Morgan, J. Peraire and M. Vahdati. Finite Element Flux-Corrected Transport (FEM-FCT) for the Euler and Navier–Stokes Equations; *Int. J. Num. Meth. Fluids* 7, 1093–1109 (1987).

Löhner, R., K. Morgan, M. Vahdati, J.P. Boris and D.L. Book. FEM-FCT: Combining Unstructured Grids with High Resolution; *Comm. Appl. Num. Meth.* 4, 717–730 (1988).

Löhner, R., J. Camberos and M. Merriam. Parallel Unstructured Grid Generation; *Comp. Meth. Appl. Mech. Eng.* 95, 343–357 (1992).

Löhner, R., R. Ramamurti and D. Martin. A Parallelizable Load Balancing Algorithm; *AIAA*-93-0061 (1993).

Löhner, R., C. Yang, J. Cebral, J.D. Baum, H. Luo, D. Pelessone and C. Charman. Fluid-Structure Interaction Using a Loose Coupling Algorithm and Adaptive Unstructured Grids; *AIAA*-95-2259 (1995).

Löhner, R., C. Yang and E. Oñate. Viscous Free Surface Hydrodynamics Using Unstructured Grids; *Proc. 22nd Symp. Naval Hydrodynamics*, Washington, DC, August (1998).

Löhner, R., C. Yang, J. Cebral, J.D. Baum, H. Luo, E. Mestreau, D. Pelessone and C. Charman. Fluid-Structure Interaction Algorithms for Rupture and Topology Change; *Proc. 1999 JSME Computational Mechanics Division Meeting*, Matsuyama, Japan, November (1999a).

Löhner, R. C. Yang, J.D. Baum, H. Luo, D. Pelessone and C. Charman. The Numerical Simulation of Strongly Unsteady Flows With Hundreds of Moving Bodies; *Int. J. Num. Meth. Fluids* 31, 113–120 (1999b).

Löhner, R., C. Yang, E. Oñate and S. Idelssohn. An Unstructured Grid-Based, Parallel Free Surface Solver; *Appl. Num. Math.* 31, 271–293 (1999c).

Löhner, R., C. Sacco, E. Oñate and S. Idelsohn. A Finite Point Method for Compressible Flow; *Int. J. Num. Meth. Eng.* 53, 1765–1779 (2002).

Löhner, R., O. Soto and C. Yang. An Adjoint-Based Design Methodology for CFD Optimization Problems; *AIAA*-03-0299 (2003).

Löhner, R., J.D. Baum and E.L. Mestreau. Advances in Adaptive Embedded Unstructured Grid Methods; *AIAA*-04-0083 (2004a).

Löhner, R., J.D. Baum, E. Mestreau, D. Sharov, C. Charman and D. Pelessone. Adaptive Embedded Unstructured Grid Methods; *Int. J. Num. Meth. Eng.* 60, 641–660 (2004b).

Löhner, R., J.D. Baum and D. Rice. Comparison of Coarse and Fine Mesh 3-D Euler Predictions for Blast Loads on Generic Building Configurations; *Proc. MABS-18 Conf.*, Bad Reichenhall, Germany, September (2004c).

Löhner, R., J. Cebral, C. Yang, J.D. Baum, E. Mestreau, C. Charman and D. Pelessone. Large-Scale Fluid Structure Interaction Simulations; *Computing in Science and Engineering (CiSE)* May/June'04, 27–37 (2004d).

Löhner, R., H. Luo, J.D. Baum and D. Rice. Selective Edge Deactivation for Unstructured Grids with Cartesian Cores; *AIAA*-05-5232 (2005).

Löhner, R., C. Yang and E. Oñate. On the Simulation of Flows with Violent Free Surface Motion; *Comp. Meth. Appl. Mech. Eng.* 195, 5597–5620 (2006).

Löhner, R., S. Appanaboyina and J. Cebral. Comparison of Body-Fitted, Embedded and Immersed Solutions for Low Reynolds-Number Flows; *AIAA*-07-1296 (2007a).

Löhner, R., J.D. Baum, E.L. Mestreau and D. Rice. Comparison of Body-Fitted, Embedded and Immersed 3-D Euler Predictions for Blast Loads on Columns; *AIAA*-07-1133 (2007b).

Löhner, R., C. Yang and E. Oñate. Simulation of Structural Response to Violent Free Surface Flows; *4th Int. Cong. Num. Meth. Eng. Appl. Sciences*, Morelia, Mexico, January (2007c).

Lomax, H., T.H. Pulliam and D.W. Zingg. *Fundamentals of Computational Fluid Dynamics;* Springer-Verlag (2001).

Long, L.N., M.M.S. Khan and H.T. Sharp. A Massively Parallel Three-Dimensional Euler/Navier–Stokes Method; *AIAA*-89-1937-CP (1989).

Loth, E. J.D. Baum and R. Löhner. Formation of Shocks Within Axisymmetric Nozzles; *AIAA J.* 30(1), 268–270 (1992).

Luo, H., J.D. Baum, R. Löhner and J. Cabello. Adaptive Edge-Based Finite Element Schemes for the Euler and Navier–Stokes Equations; *AIAA*-93-0336 (1993).

Luo, H., J.D. Baum and R. Löhner. Edge-Based Finite Element Scheme for the Euler Equations; *AIAA J.* 32(6), 1183–1190 (1994a).

Luo, H., J.D. Baum, R. Löhner and J. Cabello. Implicit Finite Element Schemes and Boundary Conditions for Compressible Flows on Unstructured Grids; *AIAA*-94-0816 (1994b).

Luo, H., J.D. Baum and R. Löhner. A Finite Volume Scheme for Hydrodynamic Free Boundary Problems on Unstructured Grids; *AIAA*-95-0668 (1995).

Luo, H., J.D. Baum and R. Löhner. A Fast, Matrix-Free Implicit Method for Compressible Flows on Unstructured Grids; *J. Comp. Phys.* 146, 664–690 (1998).

Luo, H., J.D. Baum and R. Löhner. An Accurate, Fast, Matrix-Free Implicit Method for Computing Unsteady Flows on Unstructured Grids; *AIAA*-99-0937 (1999).

Luo, H., D. Sharov, J.D. Baum and R. Löhner. A Class of Matrix-free Implicit Methods for Compressible Flows on Unstructured Grids; *First Int. Conf. on Computational Fluid Dynamics*, Kyoto, Japan, July 10–14 (2000).

Luo, H., J.D. Baum and R. Löhner. An Accurate, Fast, Matrix-Free Implicit Method for Computing Unsteady Flows on Unstructured Grids; *Comp. and Fluids* 30, 137–159 (2001).

MacCormack, R.W. The Effect of Viscosity in Hypervelocity Impact Cratering; *AIAA*-69-354 (1969).

MacCormack, R.W. A Numerical Method for Solving the Equations of Compressible Viscous Flow; *AIAA J.* 20, 1275–1281 (1982).

Marchant, M.J. and N.P. Weatherill. The Construction of Nearly Orthogonal Multiblock Grids for Compressible Flow Simulation; *Comm. Appl. Num. Meth.* 9, 567–578 (1993).

Marco, N. and F. Beux. Multilevel Optimization: Application to One-Shot Shape Optimum Design; *INRIA Rep.* 2068 (1993).

Marco, N. and A. Dervieux. Multilevel Parametrization for Aerodynamic Optimization of 3-D Shapes; *INRIA Rep.* 2949 (1996).

Marcum, D.L. Generation of Unstructured Grids for Viscous Flow Applications; *AIAA*-95-0212 (1995).

Marcum, D.L. and N.P. Weatherill. Unstructured Grid Generation Using Iterative Point Insertion and Local Reconnection; *AIAA J.* 33(9), 1619–1625 (1995).

Marroquim, R., P.R. Cavalcanti, C. Esperanca and L. Velho. Adaptive Milti-Resolution Triangulations Based on Physical Compression; *Comm. Num. Meth. Eng.* 21, 571–580 (2005).

Martin, D. and R. Löhner. An Implicit Linelet-Based Solver for Incompressible Flows; *AIAA*-92-0668 (1992).

Martin, J.C. and W.J. Moyce. An Experimental Study of the Collapse of a Liquid Column on a Rigid Horizontal Plane; *Phil. Trans. Royal Soc. London* A 244, 312–324 (1952).

Martinelli, L. and J.R. Farmer. Sailing Through the Nineties: Computational Fluid Dynamics for Ship Performance Analysis and Design; Ch. 27 in *Frontiers of Computational Fluid Dynamics* (D.A. Caughey and M.M. Hafez eds.), John Wiley & Sons (1994).

Martinelli, L. and A. Jameson. Validation of a Multigrid Method for the Reynolds Averaged Equations; *AIAA*-88-0414 (1988).

Martins, M.A.D., J.L.D. Alves and A.L.G.A. Coutinho. Parallel Edge-based Finite Element Techniques for Nonlinear Solid Mechanics; *Proc. 4th Int. Conf. on Vector and Parallel Processing*, O Porto, Portugal, June (2000).

Matus, R.J. and E.E. Bender. Application of a Direct Solver for Space-Marching Solutions; *AIAA*-90-1442 (1990).

Mavriplis, C. A Posteriori Error Estimators for Adaptive Spectral Element Techniques; *Proc. 8th GAMM Conf. Num. Meth. Fluid Mech., NNFM* 29, pp. 333–342, Vieweg (1990a).

Mavriplis, D.J. Adaptive Mesh Generation for Viscous Flows Using Delaunay Triangulation; *J. Comp. Phys.* 90, 271–291 (1990b).

Mavriplis, D.J. Turbulent Flow Calculations Using Unstructured and Adaptive Meshes; *Int. J. Num. Meth. Fluids* 13, 1131–1152 (1991a).

Mavriplis, D.J. Three-Dimensional Unstructured Multigrid for the Euler Equations; *AIAA*-91-1549-CP (1991b).

Mavriplis, C. Adaptive Mesh Strategies for the Spectral Element Method; *ICASE Rep.* 92–36 (1992).

Mavriplis, D.J. An Advancing Front Delaunay Triangulation Algorithm Designed for Robustness; *AIAA*-93-0671 (1993).

Mavriplis, D. A Unified Multigrid Solver for the Navier–Stokes Equations on Unstructured Meshes; *AIAA*-95-1666 (1995).

Mavriplis, D. A 3-D Agglomeration Multigrid Solver for the Reynolds-Averaged Navier–Stokes Equations on Unstructured Meshes; *Int. J. Num. Meth. Fluids* 23, 527–544 (1996).

Mavriplis, D.J. Adaptive Meshing Techniques for Viscous Flow Calculations on Mixed-Element Unstructured Meshes; *AIAA*-97-0857 (1997).

Mavriplis, D. and A. Jameson. Multigrid Solution of the Two-Dimensional Euler Equations on Unstructured Triangular Meshes; *AIAA*-87-0353 (1987).

Mavriplis, D.J. and A. Jameson. Multigrid Solution of the Two-Dimensional Euler Equations on Unstructured Triangular Meshes; *AIAA J.* 28(8), 1415–1425 (1990).

Mavriplis, D.J. and S. Pirzadeh. Large-Scale Parallel Unstructured Mesh Computations for Three-Dimensional High-Lift Analysis; *AIAA J. of Aircraft* 36(6), 987–998 (1999).

Mavriplis, D., M. Aftosmis and M. Berger. High Resolution Aerospace Applications Using the NASA Columbia Supercomputer; *Proc. Supercomputing'05*, Seattle, WA, November 12–18 (2005).

McGrory, W.D., R.W. Walters and R. Löhner. Three-Dimensional Space-Marching Algorithm on Unstructured Grids; *AIAA J.* 29(11), 1844–1849 (1991).

Meakin, R.L. Moving Body Overset Grid Methods for Complete Aircraft Tiltrotor Simulations; *AIAA*-93-3350-CP (1993).

Meakin, R.L. On Adaptive Refinement and Overset Structured Grids; *AIAA*-97-1858 (1997).

Meakin, R.L. and N. Suhs. Unsteady Aerodynamic Simulations of Multiple Bodies in Relative Motion; *AIAA*-89-1996 (1989).

Medic, G., B. Mohammadi and S. Moreau. Optimal Airfoil and Blade Design in Compressible and Incompressible Flows; *AIAA*-98-2898 (1998).

Mehrota, P., J. Saltz and R. Voigt (eds.). *Unstructured Scientific Computation on Scalable Multiprocessors;* MIT Press (1992).

Melton, J.E., M.J. Berger and M.J. Aftosmis. 3-D Applications of a Cartesian Grid Euler Method; *AIAA*-93-0853-CP (1993).

Merriam, M. An Efficient Advancing Front Algorithm for Delaunay Triangulation; *AIAA*-91-0792 (1991).

Merriam, M. and T. Barth. 3D CFD in a Day: The Laser Digitizer Project; *AIAA*-91-1654 (1991).

Mestreau, E., R. Löhner and S. Aita. TGV Tunnel-Entry Simulations Using a Finite Element Code with Automatic Remeshing; *AIAA*-93-0890 (1993).

Miller, K. and R.N. Miller; *SIAM J. Num. Anal.* 18, 1019 (1981).

Minami, Y. and M. Hinatsu. Multi Objective Optimization of Ship Hull Form Design by Response Surface Methodology; *Proc. 24th Symp. Naval Hydrodynamics*, Fukuoka, Japan, July (2002).

Mittal, R. and G. Iaccarino. Immersed Boundary Methods; *Annu. Rev. Fluid Mech.* 37, 239–261 (2005).

Miyata, H. and K. Gotoda. Hull Design by CAD/CFD Simulation; *Proc. 23rd Symp. Naval Hydrodynamics*, Val de Reuil, France, September (2000).

Mohammadi, B. A New Optimal Shape Design Procedure for Inviscid and Viscous Turbulent Flows; *Int. J. Num. Meth. Fluids* 25, 183–203 (1997).

Mohammadi, B. Flow Control and Shape Optimization in Aeroelastic Configurations; *AIAA*-99-0182 (1999).

Mohammadi, B. and O. Pironneau. *Applied Shape Optimization for Fluids;* Oxford University Press (2001).

Mohd-Yusof, J. Combined Immersed-Boundary/B-Spline Methods for Simulations of Flow in Complex Geometries; *CTR Annual Research Briefs*, NASA Ames Research Center/Stanford University, 317–327 (1997).

Molino, N., R. Bridson, J. Teran and R. Fedkiw. A Crystalline, Red Green Strategy for Meshing Highly Deformable Objects with Tetrahedra; pp. 103–114 in *Proc. 12th Int. Meshing Roundtable*, Santa Fe, New Mexico (2003).

Moore, G.E. Cramming More Components Onto Integrated Circuits; *Electronics* 38, 8 (1965).

Moore, G.E. A Pioneer Looks Back at Semiconductors; *IEEE Design & Test of Computers* 16(2), 8–14 (1999).

Moore, G.E. No Exponential is Forever . . . but We Can Delay 'Forever'; paper presented at the *Int. Solid State Circuits Conference (ISSCC)*, February 10 (2003).

Moos, O., F.R. Klimetzek and R. Rossmann. Bionic Optimization of Air-Guiding Systems; *SAE*-2004-01-1377 (2004).

Morgan, K., J. Probert and J. Peraire. Line Relaxation Methods for the Solution of Two-Dimensional and Three-Dimensional Compressible Flows; *AIAA*-93-3366 (1993).

Morino, H. and K. Nakahashi. Space-Marching Method on Unstructured Hybrid Grid for Supersonic Viscous Flows; *AIAA*-99-0661 (1999).

Müller, J.-D. Proven Angular Bounds and Stretched Triangulations With the Frontal Delaunay Method; *AIAA*-93-3347-CP (1993).

Müller, J., P.L. Roe and H. Deconinck. A Frontal Approach for Internal Node Generation in Delaunay Triangulations; *Int. J. Num. Meth. Eng.* 17(2), 241–256 (1993).

Nakahashi, K. FDM-FEM Zonal Approach for Viscous Flow Computations over Multiple Bodies; *AIAA*-87-0604 (1987).

Nakahashi, K. Optimum Spacing Control of the Marching Grid Generation; *AIAA*-88-0515 (1988).

Nakahashi, K. and G. S. Deiwert. A Three-Dimensional Adaptive Grid Method; *AIAA*-85-0486 (1985).

Nakahashi, K. and S. Obayashi. Viscous Flow Computations Using a Composite Grid; *AIAA-CP*-87-1128, *8th CFD Conf.*, Hawaii (1987).

Nakahashi, K. and E. Saitoh. Space-Marching Method on Unstructured Grid for Supersonic Flows with Embedded Subsonic Regions; *AIAA*-96-0418 (1996); see also *AIAA J.* 35(8), 1280–1285 (1997).

Nakahashi, K. and D. Sharov. Direct Surface Triangulation Using the Advancing Front Method; *AIAA*-95-1686-CP (1995).

Nakahashi, K., F. Togashi and D. Sharov. An Intergrid-Boundary Definition Method for Overset Unstructured Grid Approach; *AIAA*-99-3304-CP (1999); see also *AIAA J.* 38(11), 2077–2084 (2000).

Narducci, R., B. Grossman, M. Valorani, A. Dadone and R.T. Haftka. Optimization Methods for Non-Smooth or Noisy Objective Functions in Fluid Design Problems; *AIAA*-95-1648-CP (1995).

Naujoks, B., L. Willmes, W. Haase, T. Bäck and M. Schütz. Multi-Point Airfoil Optimization Using Evolution Strategies; *Proc. ECCOMAS 2000*, Barcelona (2000).

Nay, R.A. and S. Utku. An Alternative for the Finite Element Method; *Variational Methods Eng.* 1, 62–74 (1972).

Naylor, D.J. Filling Space With Tetrahedra; *Int. J. Num. Meth. Eng.* 44, 1383–1395 (1999).

Newman, J.C., W.K. Anderson and D.L. Whitfield. Multidiscipliinary Sensitivity Derivatives Using Complex Variables; *MSSU-EIRS-ERC*-98-08 (1998).

Newman, J.C., A.C. Taylor, R.W. Barnwell, P.A. Newman and G.J. Hou. Overview of Sensitivity Analysis and Shape Optimization for Complex Aerodynamic Configurations; *J. of Aircraft* 36(1), 87–96 (1999).

Nguyen, V-N., W.G. Habashi and M.V. Bhat. Vector-Parallel Gauss Elimination Solver for Large-Scale Finite Element Computational Fluid Dynamics; pp. 363–369 in *Proc. Supercomputing Symp. ’90*, Montreal, June (1990).

Nichols, B.D. and C.W. Hirt. Methods for Calculating Multi-Dimensional, Transient Free Surface Flows Past Bodies; *Proc. First Int. Conf. Num. Ship Hydrodynamics*, Gaithersburg, ML, Oct. 20–23 (1975).

Nicolaides, R.A. Deflation of Conjugate Gradients with Applications to Boundary Value Problems; *SIAM J. Num. Anal.* 24(2), 355–365 (1987).

Nielsen, E. and W. Anderson. Aerodynamic Design Optimization on Unstructured Meshes Using the Navier–Stokes Equations; *AIAA*-98-4809 (1998).

Nielsen, E. and W. Anderson. Recent Improvements in Aerodynamic Design and Optimization on Unstructured Meshes; *AIAA*-01-0596 (2001).

Nirschl, H., H.A. Dwyer and V. Denk. A Chimera Grid Scheme for the Calculation of Particle Flows; *AIAA*-94-0519 (1994).

Obayashi, S. Pareto Genetic Algorithm for Aerodynamic Design Using the Navier–Stokes Equations; pp. 245–266 in *Genetic Algorithms in Engineering and Computer Science* (D. Quagliarella *et al.* eds.), John Wiley & Sons (1998).

Obayashi, S. Pareto Solutions of Multipoint Design of Supersonic Wings Using Evolutionary Algorithms; *Proc. 5th Int. Conf. on Adaptive Computing in Design and Manufacture*, Exeter, Devon, UK, April (2002).

Obayashi, S., K. Nakahashi, A. Oyama and N. Yoshino. Design Optimization of Supersonic Wings Using Evolutionary Algorithms; *Proc. ECCOMAS 98*, John Wiley & Sons (1998).

Oden, J.T., P. Devloo and T. Strouboulis. Adaptive Finite Element Methods for the Analysis of Inviscid Compressible Flow: I. Fast Refinement/Unrefinement and Moving Mesh Methods for Unstructured Meshes; *Comp. Meth. Appl. Mech. Eng.* 59, 327–362 (1986).

Olsen, H. Unpublished Sloshing Experiments at the Technical University of Delft; Delft, The Netherlands (1970).

Olsen, H. and K.R. Johnsen. Nonlinear Sloshing in Rectangular Tanks. A Pilot Study on the Applicability of Analytical Models; *Det Norske Veritas Rep.* 74-72-S, Vol. II (1975).

Oñate, E., S. Idelsohn, O.C. Zienkiewicz and R.L. Taylor. A Finite Point Method in Computational Mechanics. Applications to Convective Transport and Fluid Flow; *Int. J. Num. Meth. Eng.* 39, 3839–3866 (1996a).

Oñate, E., S. Idelsohn, O.C. Zienkiewicz, R.L. Taylor and C. Sacco. A Stabilized Finite Point Method for Analysis of Fluid Mechanics Problems; *Comp. Meth. Appl. Mech. Eng.* 139, 315–346 (1996b).

Oran, E. and J.P. Boris. *Numerical Simulation of Reactive Flow;* Elsevier (1987).

Oran, E.S., J.P. Boris and E.F. Brown. Fluid-Dynamic Computations on a Connection Machine - Preliminary Timings and Complex Boundary Conditions; *AIAA*-90-0335 (1990).

Osher, S. and R. Fedkiw. *Level Set Methods and Dynamic Implicit Surfaces;* Springer-Verlag (2002).

Osher, S. and F. Solomon. Upwind Difference Schemes for Hyperbolic Systems of Conservation Laws; *Math. Comp.* 38, 339–374 (1982).

Othmer, C., T. Kaminski and R. Giering. Computation of Topological Sensitivities in Fluid Dynamics: Cost Function Versatility; in *ECCOMAS CFD 2006* (P. Wesseling, E. Oñate and J. Périaux eds.), Delft, The Netherlands (2006).

Overset Composite Grid and Solution Technology Symposia. www.arl.hpc.mil/Overset2002/; www.arl.hpc.mil/events/Overset2004/; www.arl.hpc.mil/events/Overset2006/.

Owen, J. *et al. Thermal Analysis Workbook*, Vol. I; NASA Marshall, Huntville, Alabama (1990).

Palmerio, B. and A. Dervieux. Application of a FEM Moving Node Adaptive Method to Accurate Shock Capturing; *Proc. First Int. Conf. on Numerical Grid Generation in CFD*, Landshut, W. Germany, July 14–17 (1986).

Palmerio, B., V. Billey, A. Dervieux and J. Periaux. Self-Adaptive Mesh Refinements and Finite Element Methods for Solving the Euler Equations; *Proc. ICFD Conf. Num. Meth. for Fluid Dynamics*, Reading, UK, March 1985.

Papay, M. and R. Walters. A general Inverse Design Procedure for Aerodynamic Bodies; *AIAA*-95-0088 (1995).

Papila, N., W. Shyy, N. Fitz-Coy and R. Haftka. Assessment of Neural Net and Polynomial-Based Techniques for Aerodynamic Applications; *AIAA*-99-3167 (1999).

Parrott, A.K. and M.A. Christie. FCT Applied to the 2-D Finite Element Solution of Tracer Transport by Single Phase Flow in a Porous Medium; *Proc. ICFD-Conf. Num. Meth. in Fluid Dyn.* (K.W. Morton and M.J. Baines eds.), Reading, Academic Press (1986).

Patankar, S.V. *Numerical Heat Transfer and Fluid Flow Computational Methods in Mechanics and Thermal Science;* Hemisphere Publishing Corporation (1980).

Patankar, N.A., P. Singh, D.D. Joseph, R. Glowinski and T.W. Pan. A New Formulation of the Distributed Lagrange Multiplier/Fictitious Domain Method for Particulate Flows; *Int. J. Multiphase Flow* 26(9), 1509–1524 (2000).

Patera, A.T. A Spectral Element Method for Fluid Dynamics: Laminar Flow in a Channel Expansion. *J. Comp. Phys.* 54, 468–488 (1984).

Patnaik, G., R.H. Guirguis, J.P. Boris and E.S. Oran. A Barely Implicit Correction for Flux-Corrected Transport; *J. Comp. Phys.* 71, 1–20 (1987).

Patnaik, G., K. Kailasanath, K.J. Laskey and E.S. Oran. Detailed Numerical Simulations of Cellular Flames; *Proc. 22nd Symposium (Int.) on Combustion,* The Combustion Institute, Pittsburgh, (1989).

Patnaik, G., K. Kailasanath and E.S. Oran. Effect of Gravity on Flame Instabilities in Premixed Gases; *AIAA J.* 29(12), 2141–2147 (1991).

Peiro, J., J. Peraire and K. Morgan. The Generation of Triangular Meshes on Surfaces; *Proc. POLYMODEL XII Conf.,* Newcastle upon Tyne, May 23–24 (1989).

Peller, N., A. LeDuc, F. Tremblay and M. Manhart. High-Order Stable Interpolations for Immersed Boundary Methods; *Int. J. Num. Meth. Fluids* 52, 1175–1193 (2006).

Pember, R.B., J.B. Bell, P. Colella, W.Y. Crutchfield and M.L. Welcome. An Adaptive Cartesian Grid Method for Unsteady Compressible Flow in Irregular Regions. *J. Comp. Phys.* 120, 278 (1995).

Peraire, J., M. Vahdati, K. Morgan and O.C. Zienkiewicz. Adaptive Remeshing for Compressible Flow Computations; *J. Comp. Phys.* 72, 449–466 (1987).

Peraire, J., J. Peiro, L. Formaggia K. Morgan and O.C. Zienkiewicz. Finite Element Euler Calculations in Three Dimensions; *Int. J. Num. Meth. Eng.* 26, 2135–2159 (1988).

Peraire, J., K. Morgan and J. Peiro. Unstructured Finite Element Mesh Generation and Adaptive Procedures for CFD; *AGARD-CP-464*, 18 (1990).

Peraire, J., J. Peiro and K. Morgan. A Three-Dimensional Finite Element Multigrid Solver for the Euler Equations; *AIAA*-92-0449 (1992a).

Peraire, J., J. Peiro and K. Morgan. Adaptive Remeshing for Three-Dimensional Compressible Flow Computations; *J. Comp. Phys.* 103, 269–285 (1992b).

Peraire, J., K. Morgan and J. Peiro. The Simulation of 3D Incompressible Flows Using Unstructured Grids; Ch. 16 in *Frontiers of Computational Fluid Dynamics* (D.A. Caughey and M.M. Hafez eds.), John Wiley & Sons (1994).

Peri, D. and E.F. Campana. Multidisciplinary Design Optimization of a Naval Surface Combatant; *J. Ship Research* 47(1), 1–12 (2003).

Periaux, J. Robust Genetic Algorithms for Optimization Problems in Aerodynamic Design; pp. 371–396 in *Genetic Algorithms in Engineering and Computer Science* (G. Winter, J. Periaux, M. Galan and P. Cuesta eds.), John Wiley & Sons (1995).

Peskin, C.S. The Immersed Boundary Method; *Acta Numerica* 11, 479–517 (2002).

Pflistch, A., M. Kleeberger and H. Küsel. On The Vertical Structure of Air Flow in The Subway New York City and Dortmund (Germany); *Proc. 4th Annual George Mason University Transport and Dispersion Modeling Workshop*, Fairfax, VA, July (2000).

Pierce, N.A. and M.B. Giles. Adjoint Recovery of Superconvergent Functionals from PDE Approximations; *SIAM Rev.*, 42(2), 247–264 (2000).

Pierce, N., M. Giles, A. Jameson and L. Martinelli. Accelerating Three-Dimensional Navier–Stokes Calculations; *AIAA*-97-1953-CP (1997).

Piessanetzky, S. *Sparse Matrix Technology;* Academic Press (1984).

Pironneau, O. On Optimum Profiles in Stokes Flow; *J. Fluid Mech.* 59, 117–128 (1973).

Pironneau, O. On Optimum Design in Fluid Mechanics; *J. Fluid Mech.* 64, 97–110 (1974).

Pironneau, O. *Optimal Shape Design for Elliptic Systems;* Springer-Verlag (1985).

Pirzadeh, S. Structured Background Grids for Generation of Unstructured Grids by Advancing Front Method; *AIAA J.* 31(2), 257–265 (1993a).

Pirzadeh, S. Unstructured Viscous Grid Generation by Advancing-Layers Method; *AIAA*-93-3453 (1993b).

Pirzadeh, S. Viscous Unstructured Three-Dimensional Grids by the Advancing-Layers Method; *AIAA*-94-0417 (1994).

Popovic, J. and H. Hoppe. Progressive Simplicial Complexes; *SIGGRAPH 1997*, 217–224 (1997).

Probert, E.J., O. Hassan, K. Morgan and J. Peraire. An Adaptive Finite Element Method for Transient Compressible Flows with Moving Boundaries; *Int. J. Num. Meth. Eng.* 32, 751–765 (1991).

Pruhs, K.R., T.F. Znati and R.G. Melhem. Dynamic Mapping of Adaptive Computations onto Linear Arrays; pp. 285–300 in *Unstructured Scientific Computation on Scalable Multiprocessors* (P. Mehrota, J. Saltz and R. Voight eds.); MIT Press (1992).

Pulliam, T.H., M. Nemec, T. Holst and D.W. Zingg. Comparison of Evolutionary (Genetic) Algorithm and Adjoint Methods for Multi-Objective Viscous Airfoil Optimizations; *AIAA*-03-0298 (2003).

Quagliarella, D. Genetic Algorithms in Computational Fluid Dynamics; pp. 417–442 in *Genetic Algorithms in Engineering and Computer Science* (G. Winter, J. Periaux, M. Galan and P. Cuesta eds.); John Wiley & Sons (1995).

Quagliarella, D. and G. Chinnici. Usage of Approximation Techniques in Evolutionary Algorithms with Application Examples to Aerodynamic Shape Design Problems; pp. 167–189 in *Evolutionary Algorithms and Intelligent Tools in Engineering Optimization* (W. Annicchiarico, J. Periaux, M. Cerrolaza and G. Winter eds.); CIMNE (2005).

Quagliarella, D. and A.D. Cioppa. Genetic Algorithms Applied to the Aerodynamic Design of Transonic Airfoils; *AIAA*-94-1896-CP (1994).

Quirk, J.J. A Cartesian Grid Approach with Hierarchical Refinement for Compressible Flows; *NASA CR-194938, ICASE Report No. 94–51* (1994).

Ramamurti, R. and R. Löhner. Simulation of Flow Past Complex Geometries Using a Parallel Implicit Incompressible Flow Solver; pp. 1049–1050 in *Proc. 11th AIAA CFD Conf.*, Orlando, FL, July (1993).

Ramamurti, R. and R. Löhner. A Parallel Implicit Incompressible Flow Solver Using Unstructured Meshes; *Computers and Fluids* 25(2), 119–132 (1996).

Rank, E., M. Schweingruber and M. Sommer. Adaptive Mesh Generation and Transformation of Triangular to Quadrilateral Meshes; *Comm. Appl. Num. Meth.* 9, 121–129 (1993).

Rannacher, R. Adaptive Galerkin Finite Element Methods for Partial Differential Equations; *J. Comp. Appl. Math.* 128(1–2), 205–233 (2001).

Rausch, R.D., J.T. Batina and H.T.Y. Yang. Three-Dimensional Time-Marching Aerolastic Analyses Using an Unstructured-Grid Euler Method; *AIAA J.* 31(9), 1626–1633 (1993).

Raymer, D.P. *Aircraft Design: A Conceptual Approach;* AIAA Education Series, New York (1999).

Raven, H.C. A Solution Method for Nonlinear Ship Wave Resistance Problem; *Doctoral Thesis*, Maritime Research Institute, Netherlands (1996).

Rebay, S. Efficient Unstructured Mesh Generation by Means of Delaunay Triangulation and Bowyer-Watson Algorithm; *J. Comp. Phys.* 106(1), 125–138 (1993).

Regnström, B, L. Broberg and L. Larsson. Ship Stern Calculations on Overlapping Composite Grids; *Proc. 23rd Symp. Naval Hydrodynamics*, Val de Reuil, France, September (2000).

Reuther, J. and A. Jameson. Aerodynamic Shape Optimization of Wing and Wing-Body Configurations Using Control Theory; *AIAA*-95-0123 (1995).

Reuther, J., A. Jameson, J. Farmer, L. Matinelli and D. Saunders. Aerodynamic Shape Optimization of Complex Aircraft Configurations via an Adjoint Formulation; *AIAA*-96-0094 (1996).

Reuther, J., A. Jameson, J.J. Alonso, M.J. Rimlinger and D. Saunders. Constrained Multipoints Aerodynamic Shape Optimization Using and Adjoint Formulation and Parallel Computers; *J. of Aircraft* 36(1), 51–74 (1999).

Rhie, C.M. A Pressure Based Navier–Stokes Solver Using the Multigrid Method; *AIAA*-86-0207 (1986).

Rice, D.L., M.E. Giltrud, J.D. Baum, H. Luo and E. Mestreau. Experimental and Numerical Investigation of Shock Diffraction About Blast Walls; *Proc. 16th Int. Symp. Military Aspects of Blast and Shocks*, Keble College, Oxford, UK, September 10–15 (2000).

Riemann, G.F.B. Über die Fortpflanzung ebener Luftwellen von Endlicher Schwingungweite; *Abhandlungen der Königlichen Gesellschaft der Wissenschaften zu Göttingen*, Vol. 8 (1860).

Rivara, M.C. Algorithms for Refining Triangular Grids Suitable for Adaptive and Multigrid Techniques; *Int. J. Num. Meth. Eng.* 21, 745–756 (1984).

Rivara, M.C. Selective Refinement/Derefinement Algorithms for Sequences of Nested Triangulations; *Int. J. Num. Meth. Eng.* 28(12), 2889–2906 (1990).

Rivara, M.C. A 3-D Refinement Algorithm Suitable for Adaptive and Multigrid Techniques; *Comm. Appl. Num. Meth.* 8, 281–290 (1992).

Rizzi, A. and L. Eriksson. Computation of Inviscid Incompressible Flow with Rotation; *J. Fluid Mech.* 153, 275–312 (1985).

Roache, P.J. *Computational Fluid Dynamics;* Hermosa Publishers, Albuquerque, NM (1982).

Robinson, M.A. Modifying an Unstructured Cartesian Grid Code to Analyze Offensive Missiles; *Rep. Pan 31941-00*, US Army Missile and Space Int. Comm., Huntsville, AL (2002).

Roe, P.L. Approximate Riemann Solvers, Parameter Vectors and Difference Schemes; *J. Comp. Phys.* 43, 357–372 (1981).

Rogers, S.E., H.V. Cao and T.Y. Su. Grid Generation for Complex High-Lift Configurations; *AIAA*-98-3011 (1998).

Roma, A.M., C.S. Peskin and M.J. Berger. An Adaptive Version of the Immersed Boundary Method; *J. Comp. Phys.* 153, 509–534 (1995).

Rostaing, N., S. Dalmas and A. Galligo. Automatic Differentiation in Odyssée; *Tellus* 45A, 558–568 (1993).

Rubbert, P. *Proc. IBM Short Course in CFD,* Monterey, CA (1988).

Ruge, J. and K. Stüben. Efficient Solution of Finite Difference and Finite Element Equations; pp.170–212 in *Multigrid Methods for Integral and Differential Equations* (D.J. Paddon and H. Holstein eds.); Clarendon Press (1985).

Saad, Y. Krylov Subspace Methods on Supercomputers; *Siam J. Sci. Stat. Comp.* 10(6), 1200–1232 (1989).

Saad, Y. *Iterative Methods for Sparse Linear Systems;* PWS Publishing, Boston (1996).

Saad, Y. *Iterative Methods for Sparse Linear Systems* (2nd edn); SIAM (2003).

Saad, Y. and M.H. Schultz. GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems; *Siam J. Sci. Stat. Comp.* 7(3), 856–869 (1986).

Sackinger, P., P. Schunk and R. Rao. A Newton-Raphson Pseudo-Solid Domain Mapping Technique for Free and Moving Boundary Problems: A Finite Element Implementation; *J. Comp. Phys.* 125, 83–103 (1996).

Sakaguchi, H. and A. Murakami. Initial Packing in Discrete Element Modeling; pp 104–106 in *Discrete Element Methods* (B.K. Cook and R.P. Jensen eds.); ASCE (2002).

Samet, H. The Quadtree and Related Hierarchical Data Structures; *Computing Surveys* 16(2), 187–285 (1984).

Sandia National Laboratories. *Proc. International Meshing Roundtable* (1992-present).

Scannapieco, A.J. and S. L. Ossakow. Nonlinear Equatorial Spread F; *Geophys. Res. Lett.* 3, 451 (1976).

Scardovelli, R. and S. Zaleski. Direct Numerical Simulation of Free-Surface and Interfacial Flow; *Annual Review of Fluid Mechanics* 31, 567–603 (1999).

Schiff, L.B. and J.L. Steger. Numerical Simulation of Steady Supersonic Viscous Flow; *AIAA*-79-0130 (1979).

Schönauer, W., K. Raith and K. Glotz. The Principle of Difference Quotients as a Key to the Self-Adaptive Solution of Nonlinear Partial Differential Equations; *Comp. Meth. Appl. Mech. Eng.* 28, 327–359 (1981).

Schlichting, H. *Boundary Layer Theory;* McGraw-Hill (1979).

Schneider, R. A Grid-Based Algorithm for the Generation of Hexahedral Element Meshes; *Engineering With Computers* 12, 168–177 (1996).

Schneider, R. An Algorithm for the Generation of Hexahedral Element Meshes Based On An Octree Technique; pp. 195–196 in *Proc. 6th Int. Meshing Roundtable* (1997).

Schwefel, H.P. *Evolution and Optimization Seeking;* John Wiley & Sons (1995).

Sedgewick, R. *Algorithms;* Addison-Wesley (1983).

Sengupta, S., J. Häuser, P.R. Eiseman and J.F. Thompson (eds.). *Proc. 2nd Int. Conf. Numerical Grid Generation in Computational Fluid Dynamics*, Pineridge Press, Swansea, Wales (1988).

Sethian, J.A. *Level Set Methods and Fast Marching Methods;* Cambridge University Press (1999).

Shapiro, R.A. and E.M. Murman. Adaptive Finite Element Methods for the Euler Equations; *AIAA*-88-0034 (1988).

Sharov, D. and K. Nakahashi. Low Speed Preconditioning and LU-SGS Scheme for 3-D Viscous Flow Computations on Unstructured Grids; *AIAA*-98-0614 (1998).

Sharov, D., H. Luo, J.D. Baum and R. Löhner. Implementation of Unstructured Grid GMRES+LU-SGS Method on Shared-Memory, Cache-Based Parallel Computers; *AIAA*-00-0927 (2000a).

Sharov, D., H. Luo, J.D. Baum and R. Löhner. Time-Accurate Implicit ALE Algorithm for Shared-Memory Parallel Computers; pp. 387–392 in *Proc. First Int. Conf. on CFD* (N. Satofuka ed., Springer-Verlag), Kyoto, Japan, July 10–14 (2000b).

Shenton, D.N. and Z.J. Cendes. Three-Dimensional Finite Element Mesh Generation Using Delaunay Tessellation; *IEEE Trans. on Magnetics,* MAG-21, 2535–2538 (1985).

Shepard, M.S. and M.K. Georges. Automatic Three-Dimensional Mesh Generation by the Finite Octree Technique; *Int. J. Num. Meth. Eng.* 32, 709–749 (1991).

Shostko, A. and R. Löhner. Three-Dimensional Parallel Unstructured Grid Generation; *Int. J. Num. Meth. Eng.* 38, 905–925 (1995).

Simon, H. Partitioning of Unstructured Problems for Parallel Processing; *NASA Ames Tech. Rep.* RNR-91-008 (1991).

Simpson, T., T. Mauery, J. Korte and F. Mistree. Comparison of Response Surface and Kriging Models for Multidisciplinary Design Optimization; *AIAA*-98-4755 (1998).

Sivier, S., E. Loth, J.D. Baum and R. Löhner. Vorticity Produced by Shock Wave Diffraction; *Shock Waves* 2, 31–41 (1992).

Sloan, S.W. and G.T. Houlsby. An Implementation of Watson's Algorithm for Computing 2-Dimensional Delaunay Triangulations; *Adv. Eng. Software* 6(4), 192–197 (1984).

Smagorinsky, J. General Circulation Experiments with the Primitive Equations, I. The Basic Experiment; *Mon. Weather Rev.*, 91, 99–164 (1963).

Sod, G. A Survey of Several Finite Difference Methods for Systems of Nonlinear Hyperbolic Conservation Laws; *J. Comp. Phys.* 27, 1–31 (1978).

Soltani, S., K. Morgan and J. Peraire. An Upwind Unstructured Grid Solution Algorithm for Compressible Flow; *Int. J. Num. Meth. Heat and Fluid Flow* 3, 283–304 (1993).

Sorensen, K.A., O. Hassan, K. Morgan and N.P. Weatherill. A Multigrid Accelerated Time-Accurate Inviscid Compressible Fluid Flow Solution Algorithm Employing Mesh Movement and Local Remeshing; *Int. J. Num. Meth. Fluids* 43(5), 517–536 (2003).

Soto, O. and R. Löhner. CFD Shape Optimization Using an Incomplete-Gradient Adjoint Formulation; *Int. J. Num. Meth. Eng.* 51, 735–753 (2001a).

Soto, O. and R. Löhner. General Methodologies for Incompressible Flow Design Problems; *AIAA*-01-1061 (2001b).

Soto, O. and R. Löhner. A Mixed Adjoint Formulation for Incompressible RANS Problems; *AIAA*-02-0451 (2002).

Soto, O., R. Löhner and J.R. Cebral. An Implicit Monolithic Time Accurate Finite Element Scheme for Incompressible Flow Problems; *AIAA*-01-2626-CP (2001).

Soto, O., R. Löhner and F. Camelli. A Linelet Preconditioner for Incompressible Flows; *Int. J. Num. Meth. Heat and Fluid Flow* 13(1), 133–147 (2003).

Soto, O., R. Löhner and C. Yang. An Adjoint-Based Design Methodology for CFD Problems; *Int. J. Num. Meth. Heat and Fluid Flow* 14(6), 734–759 (2004).

Soulaimani, A., M. Fortin, Y. Ouellet, G. Dhatt and F. Bertrand. Simple Continuous Pressure Elements for Two- and Three-Dimensional Incompressible Flows; *Comp. Meth. Appl. Mech. Eng.* 62, 47–69 (1987).

Souza, D.A.F., M.A.D. Martins and A.L.G.A. Coutinho. Edge-Based Adaptive Implicit/Explicit Finite Element Procedures for Three-Dimensional Transport Problems; *Comm. Num. Meth. Eng.* 21, 545–552 (2005).

Steger, J.L., F.C. Dougherty and J.A. Benek. A Chimera Grid Scheme; *Advances in Grid Generation* (K.N. Ghia and U. Ghia eds.); ASME FED-Vol. 5, June (1983).

Stein, K., T.E. Tezduyar and R. Benney. Automatic Mesh Update with the Solid-Extension Mesh Moving Technique; *Comp. Meth. Appl. Mech. Eng.* 193, 2019–2032 (2004).

Steinbrenner, J.P., J.R. Chawner and C.L. Fouts. A Structured Approach to Interactive Multiple Block Grid Generation; *AGARD-CP-464*, 8 (1990).

Strouboulis, T. and K.A. Haque. Recent Experiences with Error Estimation and Adaptivity; Part 1: Review of Error Estimators for Scalar Elliptic Problems; *Comp. Meth. Appl. Mech. Eng.* 97, 399–436 (1992a).

Strouboulis, T. and K.A. Haque. Recent Experiences with Error Estimation and Adaptivity; Part 2: Error Estimation for H-Adaptive Approximations on Grids of Triangles and Quadrilaterals; *Comp. Meth. Appl. Mech. Eng.* 100, 359–430 (1992b).

Strouboulis, T. and J.T. Oden. A Posteriori Error Estimation for Finite Element Approximations in Fluid Mechanics; *Comp. Meth. Appl. Mech. Eng.* 78, 201–242 (1990).

Süli, E. and P. Houston. Adjoint Error Correction for Integral Outputs; *Adaptive Finite Element Approximation of Hyperbolic Problems* 25, Springer-Verlag (2002).

Surazhsky, V. and C. Gotsman. Explicit Surface Remeshing; pp. 20–30 in *SGP '03: Proc. 2003 Eurographics/ACM SIGGRAPH Symp. on Geometry Processing (Eurographics Association)* (2003).

Sussman, M. and E. Puckett. A Coupled Level Set and Volume of Fluid Method for Computing 3D and Axisymmetric Incompressible Two-Phase Flows; *J. Comp. Phys.* 162, 301–337 (2000).

Sussman, M., P. Smereka and S. Osher. A Levelset Approach for Computing Solutions to Incompressible Two-Phase Flow; *J. Comp. Phys.* 114, 146–159 (1994).

Swanson, R.C. and E. Turkel. On Central-Difference and Upwind Schemes; *J. Comp. Phys.* 101, 297–306 (1992).

Sweby, P.K. High Resolution Schemes Using Flux Limiters for Hyperbolic Conservation Laws; *SIAM J. Num. Anal.* 21, 995–1011 (1984).

Szabo, B.A. Estimation and Control of Error Based on P-Convergence; Ch. 3 in *Accuracy Estimates and Adaptive Refinements in Finite Element Computations;* John Wiley & Sons (1986).

Taghavi, R. Automatic, Parallel and Fault Tolerant Mesh Generation from CAD; *Engineering with Computers* 12(3–4), 178–185 (1996).

Tahara, Y., E. Paterson, F. Stern and Y. Himeno. Flow- and Wave-Field Optimization of Surface Combatants Using CFD-Based Optimization Methods; *Proc. 23rd Symp. Naval Hydrodynamics*, Val de Reuil, France, September (2000).

Takamura, A., A., M. Zhu and D. Vinteler. Numerical Simulation of Pass-by Maneuver Using ALE Technique, *JSAE Annual Conf.* (Spring), Tokyo, May (2001).

Tanemura, M., T. Ogawa and N. Ogita. A New Algorithm for Three-Dimensional Voronoi Tessellation; *J. Comp. Phys.* 51, 191–207 (1983).

Taylor, C. and P. Hood. A Numerical Solution of the Navier–Stokes Equations Using the Finite Element Method. *Comp. Fluids* 1, 73–100 (1973).

Tezduyar, T.E. Stabilized Finite Element Formulations for Incompressible Flow Computations; *Advances in Applied Mechanics* 28, 1–44 (1992).

Tezduyar. T.E. and T.J.R. Hughes. Finite Element Formulations for Convection Dominated Flows with Particular Emphasis on the Compressible Euler Equations; *AIAA*-83-0125 (1983).

Tezduyar, T.E. and J. Liou. Grouped Element-by-Element Iteration Schemes for Incompressible Flow Computations; *Comp. Phys. Comm.* 53, 441–453 (1989).

Tezduyar, T.E., M. Behr, S.K. Aliabadi, S. Mittal and S.E. Ray. A New Mixed Preconditioning Method for Finite Element Computations; *Comp. Meth. Appl. Mech. Eng.* 99, 27–42 (1992a).

Tezduyar, T.E., M. Behr and J. Liou. A New Strategy for Finite Element Computations Involving Moving Boundaries and Interfaces. The Deforming-Spatial-Domain/Space-Time Procedure: I. The Concept and the Preliminary Numerical Tests; *Comp. Meth. Appl. Mech. Eng.* 94, 339–351 (1992b).

Tezduyar, T.E., M. Behr, S. Mittal and J. Liou. A New Strategy for Finite Element Computations Involving Moving Boundaries and Interfaces. The Deforming-Spatial-Domain/Space-Time Procedure: II. Computation of Free-Surface Flows, Two-Liquid Flows, and Flows with Drifting Cylinders; *Comp. Meth. Appl. Mech. Eng.* 94, 353–371 (1992c).

Tezduyar, T.E., S. Mittal, S.E. Ray and R. Shih. Incompressible Flow Computations With Stabilized Bilinear and Linear Equal-Order Interpolation Velocity-Pressure Elements; *Comp. Meth. Appl. Mech. Eng.* 95, 221–242 (1992d).

Thacker, W.C., A. Gonzalez and G.E. Putland. A Method for Automating the Construction of Irregular Computational Grids for Storm Surge Forecast Models; *J. Comp. Phys.* 37, 371–387 (1980).

Thomas, J.L. and M.D. Salas. Far Field Boundary Conditions for Transonic Lifting Solutions to the Euler Equations; *AIAA*-85-0020 (1985), also: *AIAA J.* 24(7), 1074–1080 (1986).

Thomasset, F. *Implementation of Finite Element Methods for Navier–Stokes Equations*. Springer-Verlag (1981).

Thompson, J.F. A Survey of Dynamically Adapted Grids in the Numerical Solution of Partial Differential Equations; *Appl. Num. Math.* 1, 3 (1985).

Thompson, J.F. A Composite Grid Generation Code for General 3D Regions. The EAGLE Code; *AIAA J.* 26(3), 271ff (1988).

Thompson, J.F., Z.U.A. Warsi and C.W. Mastin. *Numerical Grid Generation: Foundations and Applications;* North-Holland (1985).

Tilch, R. *PhD Thesis*, CERFACS, Toulouse, France (1991).

Tilch, R. and R. Löhner. Advances in Discrete Surface Grid Generation: Towards a Reliable Industrial Tool for CFD; *AIAA*-02-0862 (2002).

Tilch, R., A. Tabbal, M. Zhu, F. Dekker and R. Löhner. Combination of Body-Fitted and Embedded Grids for External Vehicle Aerodynamics; *AIAA*-07-1300 (2007).

Togashi, F., K. Nakahashi and Y. Ito. Flow Simulation of NAL Experimental Supersonic Airplane/Booster Separation Using Overset Unstructured Grids; *AIAA*-00-1007 (2000).

Togashi, F., R. Löhner, J.D. Baum, H. Luo and S. Jeong. Comparison of Search Algorithms for Assessing Airblast Effects; *AIAA*-05-4985 (2005).

Togashi, F., Y. Ito, K. Nakahashi and S. Obayashi. Extensions of Overset Unstructured Grids to Multiple Bodies in Contact; *J. of Aircraft* 43(1), 52–57 (2006a).

Togashi, F., Y. Ito, K. Nakahashi and S. Obayashi. Overset Unstructured Grids Method for Viscous Flow Computations; *AIAA J.* 44(7), 1617–1623 (2006b).

Toro, E.F. *Riemann Solvers and Numerical Methods for Fluid Dynamics;* Springer-Verlag (1999).

Tremel, U., K.A. Sorensen, S. Hitzel, H. Rieger, O. Hassan and N.P. Weatherill. Parallel Remeshing of Unstructured Volume Grids for CFD Applications; *Int. J. Num. Meth. Fluids* 53(8), 1361–1379 (2006).

Trottenberg, U., C.W. Oosterlee and A. Schuller. *Multigrid;* Academic Press (2001).

Tsuboi, K., K. Miyakoshi and K. Kuwahara. Incompressible Flow Simulation of Complicated Boundary Problems with Rectangular Grid System; *Theoretical and Applied Mech.* 40, 297–309 (1991).

Turek, S. *Efficient Solvers for Incompressible Flow Problems;* Springer Lecture Notes in Computational Science and Engineering 6; Springer-Verlag (1999).

Tyagi, M. and S. Acharya. Large Eddy Simulation of Turbulent Flows in Complex and Moving Rigid Geometries Using the Immersed Boundary Method; *Int. J. Num. Meth. Fluids* 48, 691–722 (2005).

Unverdi, S.O. and G. Tryggvason. A Front Tracking Method for Viscous Incompressible Flows; *J. Comp. Phys.* 100, 25–37 (1992).

Usab, W. and E.M. Murman. Embedded Mesh Solutions of the Euler Equation Using a Multiple-Grid Method; *AIAA*-83-1946-CP (1983).

van der Vorst, H. *Iterative Krylov Methods for Large Linear Systems;* Cambridge University Press (2003).

van Dyke, M. *An Album of Fluid Motion*, p. 122, Parabolic Press, Stanford, California (1989).

van Leer, B. Towards the Ultimate Conservative Scheme. II. Monotonicity and Conservation Combined in a Second Order Scheme; *J. Comp. Phys.* 14, 361–370 (1974).

van Phai, N. Automatic Mesh generation with Tetrahedron Elements; *Int. J. Num. Meth. Eng.* 18, 237–289 (1982).

vande Voorde, J., J. Vierendeels and E. Dick. Flow Simulations in Rotary Volumetric Pumps and Compressors With the Fictitious Domain Method; *J. Comp. Appl. Math.* 168, 491–499 (2004).

Venditti, D.A. and David L. Darmofal. Grid adaptation for Functional Outputs: Application to Two-Dimensional Inviscid Flows; *J. Comp. Phys.* 176(1), 40–69 (2002).

Venditti, D.A. and David L. Darmofal. Anisotropic Grid Adaptation for Functional Outputs: Application to Two-Dimensional Viscous Flows; *J. Comp. Phys.* 187(1), 22–46 (2003).

Venkatakrishnan, V. Newton Solution of Inviscid and Viscous Problems; *AIAA*-88-0413 (1988).

Venkatakrishnan, V. and D. Mavriplis. Implicit Solvers for Unstructured Meshes; *J. Comp. Phys.* 105(1), 83–91 (1993).

Venkatakrishnan, V. and D.J. Mavriplis. Implicit Method for the Computation of Unsteady Flows on Unstructured Grids; *AIAA*-95-1705-CP (1995).

Venkatakrishnan, V., H.D. Simon and T.J. Barth. A MIMD Implementation of a Parallel Euler Solver for Unstructured Grids; *NASA Ames Tech. Rep.* RNR-91-024 (1991).

Versteeg, H.K. and W. Malalasekera. *An Introduction to Computational Fluid Dynamics: The Finite Volume Method;* Addison-Wesley (1996).

Vicini, A. and D. Quagliarella. Inverse and Direct Airfoil Design Using a Multiobjective Genetic Algorithm; *AIAA J.* 35(9), 1499–1505 (1997).

Vicini, A. and D. Quagliarella. Airfoil and Wing Design Through Hybrid Optimization Strategies; *AIAA J.* 37(5), 634–641 (1999).

Vidwans, A., Y. Kallinderis and V. Venkatakrishnan. A Parallel Load Balancing Algorithm for 3-D Adaptive Unstructured Grids; *AIAA*-93-3313-CP (1993).

von Hanxleden, R. and L.R. Scott. Load Balancing on Message Passing Architectures; *J. Parallel and Distr. Comp.* 13, 312–324 (1991).

Walhorn, E. Ein Simultanes Berechnungsverfahren für Fluid- Struktur- Wechselwirkungen mit Finiten Raum- Zeit- Elementen; *PhD Thesis*, TU Braunschweig (2002).

Waltz, J. and R. Löhner. A Grid Coarsening Algorithm for Unstructured Multigrid Applications; *AIAA*-00-0925 (2000).

Wang, D.W., I.N. Katz and B.A. Szabo. H- and P-Version Finite Element Analyses of a Rhombic Plate; *Int. J. Num. Meth. Eng.* 20, 1399–1405 (1984).

Wang, D., O. Hassan, K. Morgan and N. Weatherill. Enhanced Remeshing from STL Files with Applications to Surface Grid Generation; *Comm. Num. Meth. Eng.* 23(3), 227–239 (2007).

Watson, D.F. Computing the N-Dimensional Delaunay Tessellation with Application to Voronoi Polytopes; *The Computer Journal* 24(2), 167–172 (1981).

Weatherill, N.P. Delaunay Triangulation in Computational Fluid Dynamics; *Comp. Math. Appl.* 24(5/6), 129–150 (1992).

Weatherill, N.P. The Delaunay Triangulation - From the Early Work in Princeton; Ch. 6 in *Frontiers of Computational Fluid Dynamics 1994* (D.A. Caughey and M.M. Hafez eds.); John Wiley & Sons (1994).

Weatherill, N.P. and O. Hassan. Efficient Three-Dimensional Delaunay Triangulation with Automatic Point Creation and Imposed Boundary Constraints; *Int. J. Num. Meth. Eng.* 37, 2005–2039 (1994).

Weatherill, N.P., P.R. Eiseman, J. Häuser and J.F. Thompson (eds.). *Proc. 4th Int. Conf. Numerical Grid Generation in Computational Fluid Dynamics and Related Fields*, Pineridge Press, Swansea, Wales (1993a).

Weatherill, N.P., O. Hassan, M.J. Marchant and D.L. Marcum. Adaptive Inviscid Flow Solutions for Aerospace Geometries on Efficiently Generated Unstructured Tetrahedral Meshes; *AIAA*-93-3390 (1993b).

Weatherill, N.P., O. Hassan and D.L. Marcum. Calculation of Steady Compressible Flowfields with the Finite Element Method; AIAA-93-0341 (1993c).

Wesseling, P. *Principles of Computational Fluid Dynamics;* Springer-Verlag (2001).

Wesseling, P. *An Introduction to Multigrid Methods;* Edwards (2004).

Westermann, T. Localization Schemes in 2-D Boundary-Fitted Grids; *J. Comp. Phys.* 101, 307–313 (1992).

Wey, T.C. The Application of an Unstructured Grid Based Overset Grid Scheme to Applied Aerodynamics; *Proc. 8th Int. Meshing Roundtable*, South Lake Tahoe, October (1999).

Whirley, R.G. and J.O. Hallquist. DYNA3D, A Nonlinear Explicit, Three-Dimensional Finite Element Code for Solid and Structural Mechanics - User Manual; *UCRL-MA*-107254, Rev. 1, (1993).

Whitaker, D.L., B. Grossman and R. Löhner. Two-Dimensional Euler Computations on a Triangular Mesh Using an Upwind, Finite-Volume Scheme; *AIAA*-89-0365 (1989).

Wigton, L.B., N.J. Yu and D.P. Young. GMRES Acceleration of Computational Fluid Dynamics Codes; *AIAA*-85-1494-CP (1985).

Williams, J.W.J. Heapsort; *Comm. ACM* 7, 347–348 (1964).

Williams, D. Performance of Dynamic Load Balancing Algorithms for Unstructured Grid Calculations; *CalTech Rep.* C3P913 (1990).

Winsor, N.K., S.A. Goldstein and R. Löhner. Numerical Modelling of Interior Ballistics from Initiation Through Projectile Launch; *Proc. 42nd Aeroballistic Range Association Meeting*, Adelaide, Australia, October 22–25, (1991).

Woan, C.-J. Unstructured Surface Grid Generation on Unstructured Quilt of Patches; *AIAA*-95-2202 (1995).

Woodward, P. and P. Colella. The Numerical Simulation of Two-Dimensional Fluid Flow with Strong Shocks; *J. Comp. Phys.* 54, 115–173 (1984).

Wu, J., J.Z. Zhu, J. Szmelter and O.C. Zienkiewicz. Error Estimation and Adaptivity in Navier–Stokes Incompressible Flows; *Comp. Mech.* 6, 259–270 (1990).

Yabe, T. Universal Solver CIP for Solid, Liquid and Gas; *CFD Review* 3 (1997).

Yabe, T. and T. Aoki. A Universal Solver for Hyperbolic Equations by Cubic-Polynomial Interpolation; *Comp. Phys. Comm.* 66, 219–242 (1991).

Yamamoto, K. and O. Inoue. Applications of Genetic Algorithm to Aerodynamic Shape Optimization; *AIAA*-95-1650-CP (1995).

Yang, J. and E. Balaras. An Embedded-Boundary Formulation for Large-Eddy Simulation of Turbulent Flows Interacting with Moving Boundaries; *J. Comp. Phys.* 215, 12–40 (2006).

Yang, C. and R. Löhner. Fully Nonlinear Ship Wave Calculation Using Unstructured Grids and Parallel Computing; pp. 125–150 in *Proc. 3rd Osaka Colloquium on Advanced CFD Applications to Ship Flow and Hull Form Design*, Osaka, Japan, May (1998).

Yang, C. and R. Löhner. Calculation of Ship Sinkage and Trim Using a Finite Element Method and Unstructured Grids; *Int. J. CFD* 16(3), 217–227 (2002).

Yang, C. and R. Löhner. Prediction of Flows Over an Axisymmetric Body with Appendages; *Proc 8th Int. Conf. on Numerical Ship Hydrodynamcis*, Busan, Korea, September (2003).

Yang, C. and R. Löhner. H_2O: Hierarchical Hydrodynamic Optimization; *Proc. 25th Symp. on Naval Hydrodynamics*, St. John's, Newfoundland and Labrador, Canada, August (2004).

Ye, T., R. Mittal, H.S. Udaykumar and W. Shyy. An Accurate Cartesian Grid Method for Viscous Incompressible Flows With Complex Immersed Boundaries; *J. Comp. Phys.* 156, 209–240 (1999).

Yee, H.C. Building Blocks for Reliable Complex Nonlinear Numerical Simulations; in *Turbulent Flow Computation* (D. Drikakis and B. Geurts eds.); Kluwer Academic (2001).

Yee, H.C. and P.K. Sweby. Global Asymptotic Behaviour of Iterative Implicit Schemes; *Int. J. Bifurcation and Chaos* 4(6), 1579–1611 (1994).

Yee, H.C., P.K. Sweby and D.F. Griffiths. Dynamical Approach Study of Spurious Steady-State Numerical Solutions for Nonlinear Differential Equations, Part I: The Dynamics of Time Discretizations and its Implications for Algorithm Development in Computational Fluid Dynamics; *J. Comp. Phys.* 97, 249–310 (1991).

Yerry, M.A. and M.S. Shepard. Automatic Three-Dimensional Mesh Generation by the Modified-Octree Technique; *Int. J. Num. Meth. Eng.* 20, 1965–1990 (1984).

Zalesak, S.T. Fully Multidimensional Flux-Corrected Transport Algorithm for Fluids; *J. Comp. Phys.* 31, 335–362 (1979).

Zalesak, S.T. *PhD Thesis*, George Mason University (2005).

Zalesak, S.T. and R. Löhner. Minimizing Numerical Dissipation in Modern Shock Capturing Schemes; pp. 1205–1211 in *Proc. 7th Int. Conf. Finite Elements in Flow Problems* (T.J. Chung and G. Karr eds.), Huntsville, AL (1989).

Zhmakin, A.I. and A.A. Fursenko. A Class of Monotonic Shock-Capturing Difference Schemes; *NRL Memo. Rep.* 4567, (1981).

Zhu, Z.W. and Y.Y. Chan. A New Genetic Algorithm for Aerodynamic Design Based on Geometric Concept; *AIAA*-98-2900 (1998).

Zhu, J.Z. and O.C. Zienkiewicz. A Simple Error Estimator and Adaptive Procedure for Practical Engineering Analysis; *Int. J. Num. Meth. Eng.* 24(3), 337–357 (1987).

Zhu, M., T. Fusegi, A. Tabbal, E. Mestreau, H. Malanda, D. Vinteler, A. Goto and M. Nohmi. A Tetrahedral Finite-Element Method for Fast 3-D Numerical Modeling and Entire Simulation of Unstready Turbulent Flow in a Mixed-Flow Pump; *Proc. FEDSM'98, ASME Fluids Engineering Division Summer Meeting*, FEDSM98-4879, June 21–25, Washington, DC (1998).

Zhu, J.Z., O.C. Zienkiewicz, E. Hinton and J. Wu. A New Approach to the Development of Automatic Quadrilateral Grid Generation; *Int. J. Num. Meth. Eng.* 32, 849–866 (1991).

Zienkiewicz, O.C. *The Finite Element Method;* McGraw-Hill (1991).

Zienkiewicz, O.C. and R. Löhner. Accelerated 'Relaxation' or Direct Solution? Future Prospects for the FEM; *Int. J. Num. Meth. Eng.* 21, 1–11, (1985).

Zienkiewicz, O.C. and K. Morgan. *Finite Elements and Approximation;* John Wiley & Sons (1983).

Zienkiewicz, O.C. and D.V. Phillips. An Automatic Mesh Generation Scheme for Plane and Curved Surfaces by Isoparametric Co-ordinates; *Int. J. Num. Meth. Eng.* 3, 519–528 (1971).

Zienkiewicz, O.C. and R. Taylor. *The Finite Element Method;* McGraw-Hill (1988).

Zienkiewicz, O.C., J.P. de S.R. Gago and D.W. Kelly. The Hierarchical Concept in Finite Element Analysis; *Comp. Struct.* 16, 53–65 (1983).

Zienkiewicz, O.C., Y.C. Liu and G.C. Huang. Error Estimation and Adaptivity in Flow Formulation of Forming Processes; *Int. J. Num. Meth. Eng.* 25(1), 23–42 (1988).

Zurmühl, R. *Matrizen und Ihre Technischen Anwendungen;* Springer-Verlag (1964).

INDEX